Microprocessor Logic Design

The Flowchart Method

Nick Tredennick

Microprocessor Logic Design

Tredennick

Digital Press

"It was a delight to read **Microprocessor Logic Design** . . . It is an excellent book on 'how to design a single-chip microprocessor' and the contents reflect the author's experience at two organizations on this subject, and he has done a splendid job . . . It is a good reference and text book for both the practicing and the budding professionals . . . The book de-mystifies some of the 'black-magic' practiced by the microprocessor design 'gurus'."
Ramautar Sharma,
AT&T

Nick Tredennick has been a Research Staff Member at the IBM Thomas J. Watson Research Center for the past six years. He is the designer of the logic and microcode of the IBM Micro/370 microprocessor and the Motorola MC68000 microprocessor and has taught in the Electrical Engineering and Computer Science Department at the University of California, Berkeley. He received his Ph.D. from the University of Texas, Austin in 1976.

Here are practical insights into the design of commercial-grade microprocessors—from a leading expert in the field!

Here is a clear, authoritative presentation of an industrial logic design method for single-chip microprocessors—the Flowchart Method. Drawing on his design experiences with the Motorola MC68000 and the IBM Micro/370, Nick Tredennick presents an organized way—but one complicated by practical considerations and "dirty reality"—to design logic. The descriptions move from Level 1 Flowcharts to Level 2 Flowcharts with tips throughout on how to avoid inevitable pitfalls and make the process run smoothly.

Here's what makes this book particularly useful:

- Over 190 illustrations highlight the design concepts.
- Helpful design examples give a better understanding of problems and ways to solve them successfully.
- The author's experience as both a practicing engineer and a teacher makes the material clear and easy to follow.
- A one-semester microprocessor design project with lesson plans.

Microprocessor Logic Design is intended for:

- Industrial chip designers and design engineers working in logic design—as an excellent "how-to" reference.
- Advanced undergraduate and graduate students in VLSI design—as a practical, proven approach.

Microprocessor Logic Design

The Flowchart Method

Microprocessor

Logic Design

The Flowchart Method

Nick Tredennick
IBM Research Staff

digital
Digital Press

Copyright © 1987 by Digital Equipment Corporation.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission of the publisher.

9 8 7 6 5 4 3 2 1

Printed in the United States of America.

Order number EY-6707E-DP.

Book and jacket design by Sandra Calef.
Diagrams designed by Carol Keller and produced by ANCO/Boston.
Composed in Mergenthaler Univers by DEKR Corporation.
Printed and bound by Murray Printing Company.

Chapter 3 is adapted from Nick Tredennick, "How to Flowchart for Hardware," *Computer*, vol. 14 (December 1981), pp. 87ff.; all rights reserved, © 1981 by the Institute of Electrical and Electronics Engineers. Appendix A appeared as Nick Tredennick, "The Impact of VLSI on Microprogramming," in *MICRO 19: The 19th Annual Workshop on Microprogramming, 1986*, pp. 2–5; all rights reserved, © 1986 by the Institute of Electrical and Electronics Engineers. Both appear herein by permission of the IEEE. Appendix E is adapted from a report prepared by the International Business Machines Corporation and is printed by permission of IBM.

Burroughs B1700 is a trademark of Unisys; Cray is a trademark of Cray Research, Inc.; IBM System/360 and IBM System/370 are trademarks of International Business Machines Corporation; iAPX286, 8048, 8086, 8088, and 80386 are trademarks of Intel Corp.; MC68000, MC68008, MC68010, MC68020, MC68030, M6800, M6801, and M6806 are trademarks of Motorola, Inc.; VAX is a trademark of Digital Equipment Corporation; Z8000 is a trademark of Zilog, Inc.

Library of Congress Cataloging-in-Publication Data

Tredennick, Nick.
Microprocessor Logic Design.

Bibliography: p.
1. Microprocessors—Design and construction.
2. Logic design. I. Title.
TK7895.M5T74 1987 621.395 ·86-29382

Preface

My objective in this book is to present a computer design method called the flowchart method. I do not survey computer designs or computer design methods. I do describe the hardware in a single-chip microprocessor, but I believe that the concepts apply to computers in general. The examples are from my design experiences with the IBM Micro/370 and the Motorola MC68000 design teams.

This book is for graduate-level electrical engineering or computer engineering students or for practicing computer designers. I assume the reader knows basic logic design, Karnaugh maps, and Boolean algebra.

The book is organized so that a semester-long design project can be undertaken in parallel. I introduce the design method using a simplified processor example, then add details, in the order a computer designer must deal with them, to design a single-chip microprocessor.

Nick Tredennick
January 5, 1987

Contents

Chapter 1 **Here's the Deal** 1
Why I Think This Description Is Important 2
Using Computers 2
The Right Structures 4

Chapter 2 **Defining a Microprocessor** 5
Microprocessor Operations Overview 8
Instruction 10

Chapter 3 **Hardware Flowcharts** 16
Prerequisites 18
Illustrated Flowchart Method Overview 19
Flowchart Objectives 23
Making a Flowchart 25
Level 1 Flowcharts 30
Level 2 Flowcharts 32
Doing Level 1 Flowcharts 36
Doing Level 2 Flowcharts 39

Chapter 4 **Implementing from Flowcharts** 50
Relationship between Flowcharts and Hardware 55
Sample Design Chronology 57
Sample Implementation Procedure 57
State Sequencer 72
Summary 76

Chapter 5 **How a Microprocessor Works** 77
Internal Clocking **84**
Timing between the Execution Unit and the External Bus **85**
Exceptions **91**
Control Store Address Selection **100**
Control Store **104**
Nanoword Decoder **105**
Communication between Execution Unit and State Sequencer **114**
Communication between Bus Controller and State Sequencer **116**
Microcode **117**
Bus Interface **123**
Mode Control **124**

Chapter 6 **The IBM Micro/370 Microprocessor** 126
Execution Unit **127**
Control Store Organization **130**
Decoders **133**
Next Address Control **133**
Interface to Bus Controller **133**
Execution Overlap **140**
Prefetching **140**
Clocking and Timing **148**
Bus Sense Amp Control **148**
Shifter and Shifter Control **151**
References **154**

Chapter 7 **Hardware Flowcharts for Micro/370** 155
Practice Level 2 Flowcharts **156**
Level 1 Flowcharts **158**
Official Level 2 Flowcharts **158**
Sample Instructions **160**

Chapter 8 **Implementing Micro/370 from Flowcharts** 190
Implementation **194**

Chapter 9 **VLSI Design Method(ologie)s** 221
Method A: The Machine Partition Method **222**
Method B: The Commercial Microprocessor Method **225**
Method C: The Logic Replacement Method **231**
Summary of Methods **232**
Contrasting Folklore with Reality **233**
Conclusion **237**
References **237**

Epilogue **A Final Word by T. A. Welch** 239
What Did He Say? **240**
What Did He Miss? **240**
Alternative Processor Styles **242**
Design Process **243**
Conclusions **244**

Appendix A **The Cultures of Microprogramming** 245

Appendix B **Design Chronology** 255

Appendix C **IBM/370 Architecture Notes** 269

Appendix D **The Micro/370 Flowcharts** 275

Appendix E **IBM Micro/370 Microprocessor General Information** 301

Appendix F **A One-Semester Design Project** 328

Appendix G **Glossary** 341

Index 359

CHAPTER

1

Here's the Deal

This book is for graduate level electrical engineering or computer engineering students. "Graduate level" means I assume you are proficient at, not just knowledgeable of, basic logic design. You can reduce a five-variable Karnaugh map to minimized logic in a few minutes. You can "read" multilevel NAND-NAND logic diagrams for output functions. And you can look at the logic diagram of a simple machine and see how it works. I would like you to read this book and somehow go beyond a procedure. Steal ideas. I describe the Micro/370 microprocessor in detail, but I am really talking about design ideas, using Micro/370 as an example. I want you to add these ideas to your design repertoire. Make them the stepping-off point to your own design experiences.

I present an industrial logic design method for single-chip microprocessors, called the flowchart method. I do this using a real example. The case study is Micro/370—a single-chip System/370 microprocessor designed using the flowchart method. Micro/370 consists of about two hundred thousand transistors (sites). I wrote this text as I did the logic for Micro/370. I also used the flowchart method when designing the logic for the Motorola MC68000.

Books describe methods as if they are step by step in practice. But methods are not step by step. There are always problems. Students lose confidence when they are unable to apply a method as cleanly as it is described. I present the flowchart method both ways—the tutorial and the dirty reality. I discuss mistakes in this book because mistakes are an integral part of what you do.

I intentionally repeat things from chapter to chapter; I do this for emphasis and to gradually introduce detail.

Why I Think This Description Is Important

This description tells how an engineer actually works. If you are a new logic designer, think of this as a way to get started, an organized way to develop your own style. This is a documented industrial logic design method, which means I wrote what I think you should do, in detail, to design the logic of a microprocessor.

The problem with many texts is that we lie about details. We are sloppy in areas that are not our primary concern. Academics are method fanatics. Practitioners are solution fanatics. In school, we glorify the methods and lie about the sophisticated problems we solved. In industry, we glorify the problems and lie about the sophisticated methods we used. Each side loses credibility the minute one side reads the other's literature. The academic knows that the practitioner's "method" is (ugh!) arbitrary, just as the practitioner knows the academic's "solution" is (ugh!) not applicable. Because we oversell method and solution, it takes too long to figure out what really works. This book puts what I think really works for microprocessor logic design in one place.

Using Computers

Eventually, you will enter your design into computer files. Lots of people have tried to make this part "easier." These people are called design automation (DA) experts. Designers work on designs, and DA people work on automating design. As a designer, my view is that DA should *support* design, not *be* design. After I have designed something, I think, "Boy, it would be nice if this part (of the way I design) were automated in this particular way." But sometimes I think DA people automate things and

want designers to design in terms of inputs and outputs to their design tools. You may feel frustrated when you ask a DA person, "Why doesn't your program let me do this?" and the response is, "Why are you doing it *that* way?"

Imagine this: You start a new job at a company, and on day 2 they say, "Here's where you enter your logic—in our Humongous Design System (HDS)."

Surprised, you say, "But I've only had eight courses in digital design principles. How do you actually *design* a microprocessor? I mean . . . come up with the logic for something that complicated . . . in an organized way?"

"Well, you partition the problem into manageable pieces," they reply.

Stubbornly you ask, "But how do you know what the right pieces are?"

Perspiration is forming on your brow. They've found you out: You were supposed to have learned this in one of those courses. But your host merely replies, "Oh, that's easy, you just piece the logic together from the structures that are good for this technology. They're right here. See, here's a sixteen-way NOR, and here's a three-input NAND and . . ."

You don't hear much of the speech. You have no choice: You must use HDS because it automatically verifies your logic; selects, places, and wires the circuits; and generates test patterns using a fault model that has been accepted companywide. Besides, output from HDS is the only output manufacturing will accept. Period.

In a case like this, designers start thinking of solving problems in terms of how to express the solution in a particular notation. They structure the solution out of only the conceptual constructs supported by that notation. (If the tool does not support pass gates, guess what—no pass gates in the design.) In this way, DA becomes design.

This book is partly in response to the growing presumption that computers are an essential part of logic design. They are not. I want to describe here the essence of a logic design method.

Computers are not an essential part of this flowchart method. I think of computers as an expensive and awkward alternative to pencil and paper. Even so, I don't think *all* DA is bad. When DA's good, it's very, very good, but when it's bad, it's horrid.

The Right Structures

The key to performance in microprocessor design is finding the right conceptual structures. How do you best represent the concepts present in the architecture document? If you are constrained by notation, by the contents of a circuit library, or by the fact that wiring must be on a grid, then you lose a lot of performance. (I make this flat statement with no proof.) I believe that basing your design on the right conceptual structures—ones that make the design "flow"—is the key to high performance. I do not believe it matters (in performance) whether these structures are implemented in programmable logic arrays (PLAs), read-only memory (ROM), or random logic. ROM- or PLA-based logic is certainly easier to change physically, but neither is inherently faster or slower than random logic.

You will frequently encounter statements such as, "Microcode leads to slower control paths and adds to interpretive overhead," and "Hardwired control provides for the fastest possible operation." These statements are not true. What *is* true is that microcoded solutions tend to be used for interpretive structures, and interpretive structures are slower. For those who believe "microcoded" means "interpretive," think of what I am discussing as microcoded implementations that are not interpretive (a seeming contradiction).

Speed comes from using the right logical structures for the job. Structured garbage logic is still garbage logic. How do you find the right logical structures? Use the flowchart method. It gives you a framework (notation and procedure) that organizes design details so that you can see logical patterns. After that, everything depends on how good you are at logic design.

Chapter 8 shows you where the flowchart method fits in relation to other methods of logic design. I prefer microcoded designs. (See Appendix A for my definition of microcoded. To me, the word "microcode" is interchangeable with "microprogram.") The examples I use, real and contrived, are mainly microcoded ones.

C H A P T E R

2

Defining a Microprocessor

A microprocessor is a computer's central processing unit (CPU) implemented on a few (say, fewer than four) silicon chips. The processor has two parts: a control part and a data part. The control part says what to do, and the data part does it. The control part decodes instructions and guides the processor through its internal states. The data part (or execution unit) contains the registers, arithmetic units, shifter, and other pieces that directly store or manipulate data. The control part directs operations in the execution unit. It consists of the clock-phase generators, bus controller, and processor controller. The processor controller consists of a control store (with all the microcode), state sequencer, instruction decoders, and control word decoder. See figure 2.1.

A single-chip microprocessor is a silicon chip containing all (and only) the parts of a CPU. The chip must include all the parts mentioned above (clock-phase generators, bus controller, processor controller, and execution unit) to be a single-chip microprocessor. Otherwise, it is one chip of a multichip microprocessor. Figure 2.1 is a block diagram of a microcoded implementation of a single-chip microprocessor. Figure 2.2 is a block diagram of a PLA implementation of a

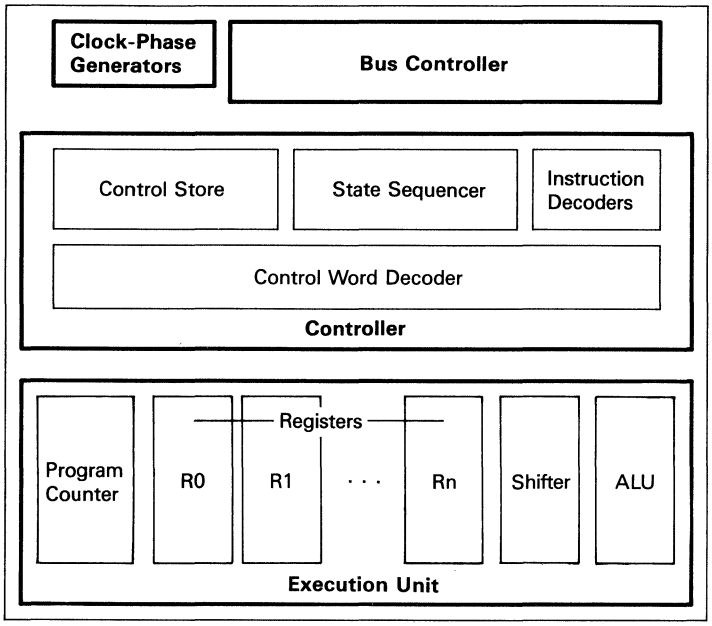


Figure 2.1 Microprocessor (microcoded implementation)

single-chip microprocessor. Figure 2.3 is a block diagram of a random logic implementation of a single-chip microprocessor. From here on, when I use the term "microprocessor," I mean single-chip microprocessor unless I say otherwise. Since I prefer microcoded implementations, I will use those as examples from now on.

"But," you say, "I want to design a high-performance microprocessor, so I want to know how to do a random logic implementation. Your book will do me no good." Our technical folklore says that random logic implementation is faster. That is not necessarily so. If the random logic implementation is faster, it is not because it is done in random logic. Figures 2.1, 2.2, and 2.3, for example, have the same execution unit (exactly what I expect if they implement the same architecture). Where is the critical path? Suppose it's in the execution unit. (A common critical path in an execution unit is the path from a register, through the arithmetic and logic unit [ALU], and into an ALU condition code register.) If so, all three implementations perform equally.

There are many microprocessors commercially available today. You can't tell which are microcoded, PLA, and random logic im-

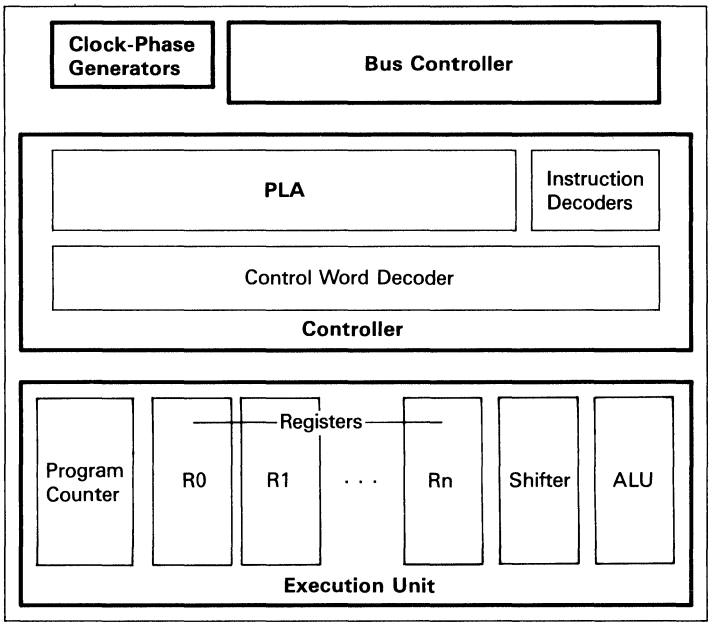


Figure 2.2 Microprocessor (PLA implementation)

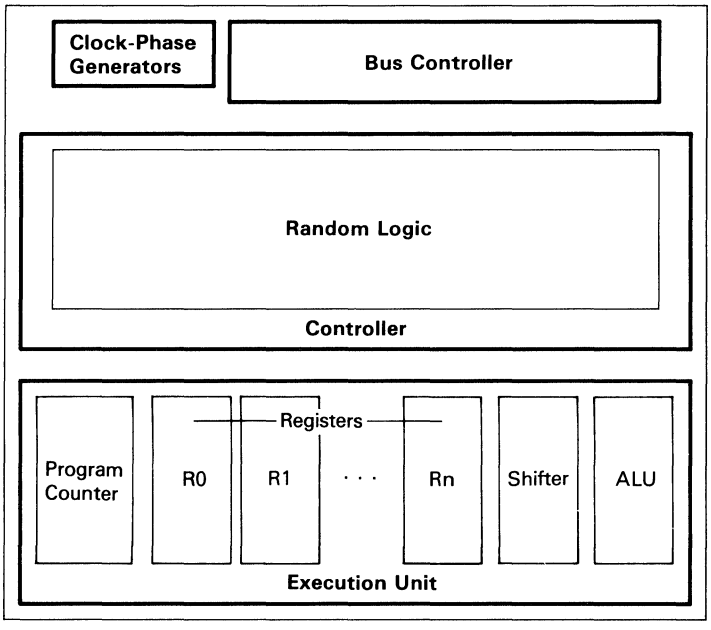


Figure 2.3 Microprocessor (random logic implementation)

plementations based on their performance. Differences in architecture swamp differences in implementation (and make comparisons unsound). The Motorola MC68000 family, which are microcoded designs, are among the fastest microprocessors available.

Why am I harping on this? Because I think that what the implementation looks like (microcoded, PLA, or random logic) when it is done is more nearly related to design method than to performance (or any other input constraint). What you do is start with a specification and design a computer to some goal—size, cost, performance. Don't worry about the form of the implementation (yet). I'll show you how to develop your own design method. Begin by building your method from the things that matter—the specification and the goals. I'll show you how to build a fast microprocessor (or a cheap one or something in-between), and I'll do it with microcoded examples.

Microprocessor Operations Overview

This is an operational overview of the microcoded microprocessor in figure 2.4. Instruction decoders look at the instruction bit pattern to decide which control word sequence in the control store is appropriate. The instruction decoders send the address of the control word sequence to the control store. The control store contains the control word sequences for all the instructions. The state sequencer steps the control store through each control word in the sequence for the instruction. The control word decoder transforms each of the control words into specific control signals for each execution unit element. The execution unit contains the resources for holding and manipulating data. Execution unit pieces (elements) are connected by one or more common internal buses. Transfers between execution unit elements are controlled by the control words. Transfers between the microprocessor internals and the external world (the world beyond the pads) are controlled by the bus controller. (There is a simple connection from the Data In/Out register to bus transceivers connected directly to the pads. Similarly, the Address Out [AO] buffer in figure 2.4 goes to drivers connected directly to the pads.) The bus controller responds to commands imbedded in the control words. It runs the external bus protocols that result in instruction fetches (for the instruction decoders) and in operand loads and stores (for the execution unit).

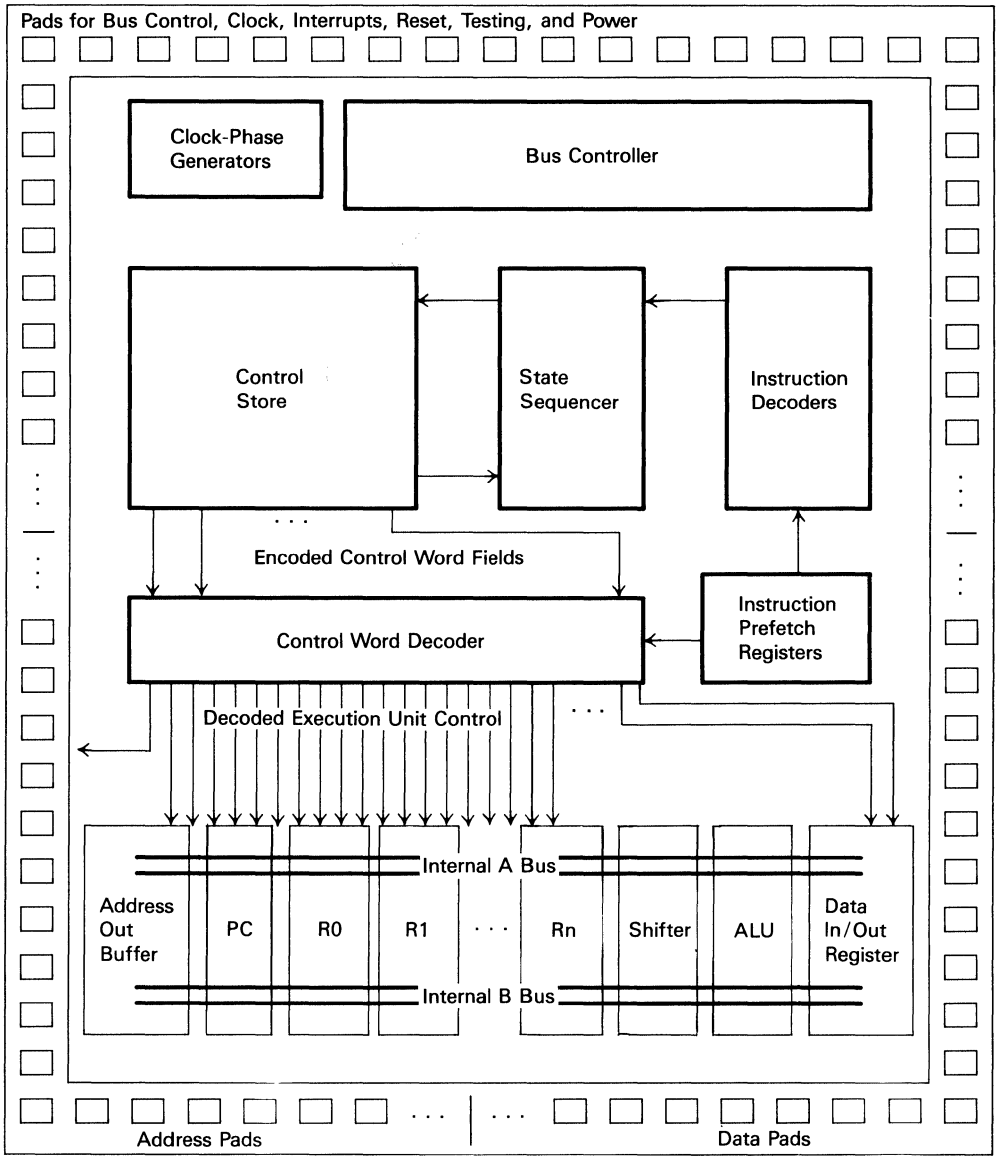


Figure 2.4 Microprocessor chip, with more detail (microcoded implementation)

Instruction

Figure 2.5 is an example of an instruction in a typical instruction reference manual. In this example, 5A (hex) is the operation (op) code, the R1 and B2 fields designate registers, and the D2 field holds a displacement. I describe how the microprocessor executes the instruction. (I assume a microcoded processor controller.) Instructions can be either one or two halfwords long (depending on whether they include a displacement). A single halfword instruction would have a different op code from the instruction example in figure 2.5 and would have no D2 halfword.

The microprocessor fetches the instruction. The first byte of the instruction gives it away as an ADD instruction. The first instruction byte drives an instruction decoder, whose output is the control store address of the control word sequence for the ADD instruction. The control word sequence for the ADD instruction knows the ADD instruction format. The control word sequence directs the execution unit to fetch the operands, add them, and store the result. First, the control word sequence fetches the rest of the instruction (in this case, three more bytes). Then it finds the operands. It finds the first operand in the register designated by the R1 field. It finds the second operand by adding the contents of the register designated by the B2 field to the displacement of the D2 field. The control word sequence calculates the address, puts the address on the pads (the external bus), and captures the data returning from memory. It adds the operands and stores the result in the register designated by the R1 field.

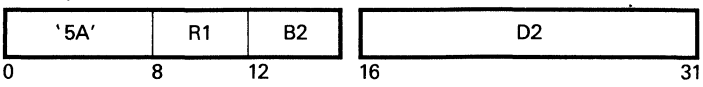
Here are the steps for the ADD instruction:

1. Fetch the first instruction halfword.
2. Find the ADD control word sequence.
3. Fetch the remaining instruction halfword.
4. Calculate the operand address.
5. Fetch the operand.
6. Add.
7. Store the answer.

It isn't quite that simple. This works for one instruction, but you must be able to execute a program (sequence of instructions). How do you get to the next instruction? How did you get here from the last one? The processor controller does this. One way to execute a sequence of instructions is to have the current instruction fetch and decode the next instruction. In a micro-

ADD

A R1,D2 (B2)



The second operand is added to the first operand, and the sum is placed in the first-operand location. The operands and the sum are treated as 16-bit signed binary integers. The first operand is in the register specified by the R1 field. The second operand is in memory. The address of the second operand is formed by adding the displacement specified by the D2 field to the contents of the base register specified by the B2 field.
An overflow causes a program interruption when the fixed-point overflow mask bit is 1.

Resulting Condition Code	Program Exceptions
0 Sum is zero	Access (fetch)
1 Sum is less than zero	Fixed-point overflow
2 Sum is greater than zero	
3 Overflow	

Figure 2.5 The ADD instruction

programmed controller, this is necessary to find the location of the control word sequence to execute the next instruction.

Assume that you have just begun execution of the ADD instruction. Here are the steps for the instruction:

1. Fetch the remaining instruction halfword.
2. Calculate the operand address.
3. Fetch the operand.
4. Add.
5. Store the answer.
6. Update the program counter (PC).
7. Fetch the first halfword of the next instruction.
8. Find the address of the next instruction's control word sequence.
9. Branch to the next instruction's control word sequence.

The steps in this sequence have been renumbered from the previous list of steps to reflect a change in instruction execution strategy. The first two steps of the initial sequence became the last four steps of the current sequence. Instead of each instruction being an independent sequence, as it is in the first set of steps, each instruction connects to the next instruction by doing its fetch and decode. These steps can execute a stream of ADD instructions. If you have a series of ADD instructions, you would execute the above steps multiple times. The first five steps do

the ADD instruction, and the last four steps connect it to the instruction stream.

A "step" is not a control word (this isn't really the control word sequence); it's only what the control word sequence must do. The control word sequence defines a series of states, and it may take several states to do each of the steps for the ADD instruction. (In a microcoded implementation, a state corresponds to one control word.)

Assume that the microprocessor in figure 2.4 has a 16-bit external data bus and 16-bit internal data buses (along with a 16-bit ALU). The following steps execute the ADD instruction:

1. Fetch the remaining instruction halfwords.
   One state to fetch the second halfword of the ADD instruction.
2. Calculate the operand address.
   One state to add the D2 displacement and the contents of the B2 register.
3. Fetch the operand.
   One state to fetch the data halfword (put the address on the pads and wait for the operand halfword).
4. Add.
   One state to add the operands.
5. Store the answer.
   One state to store the result in the R1 register.
6. Update the PC.
   One state to increment the PC.
   One state to save the incremented value.
7. Fetch the first halfword of the next instruction.
   One state to put the PC value on the pads and wait for the first half of the next instruction.
8. Find the address of the next instruction's control word sequence.
   One state to put the next instruction into the instruction decoder.
9. Branch to the next instruction's control word sequence.
   Zero states—this step is accomplished as a part of the previous step.

For a halfword (16-bit) external data bus and halfword (16-bit) internal buses, the sequence is nine states. How does this compare with the execution time (in states) for a commercial

microprocessor? The Motorola MC68000 has an internal structure similar to figure 2.4 with 16-bit internal buses (and a 16-bit ALU). It has a halfword external data bus. Its control word sequence for the memory-to-register ADD instruction is five states. "What happened? Didn't those guys have to do everything?" They did everything. My description is simplified, as it doesn't account for concurrent actions.

In this chapter, I have begun to tell you how a microprocessor works by defining one. I described how the microprocessor runs an instruction—that is, what's going on inside the chip. In chapter 5, I will continue in the same fashion. I repeat the steps in the execution of an instruction but add more detail each time. More detailed explanations of what is happening require more details about how the microprocessor works. As I add detail to the explanation, problems keep popping up. Solving each problem requires more information about how a particular part of the microprocessor works. Here are the parts in the order they are explained in chapter 5.

The upper left corner of the block diagram in figure 2.6 designates the clock-phase generators. They use the externally supplied clock signal to generate clock phases required by the rest of the chip. Both Micro/370 and the MC68000 use a four-phase clocking scheme.

Power-on reset and interrupts are next (top center to right in figure 2.6). Any microprocessor has to have power-on reset circuitry so it will do something predictable when you turn on the power. Interrupts provide a way for devices outside the microprocessor to get the microprocessor's attention. One type of interrupt tells the microprocessor when a device needs service (for example, keyboard service, display buffer update or more lines to print). Another type of interrupt informs the (on-chip) bus controller that something is wrong on the external bus and the current bus access will not complete (for instance, a bus error or page fault).

Interrupts lead naturally to the next state control (a part of the processor controller's state sequencer). Normally, the microprocessor is just running a user's program. An interrupt comes in and changes what the processor controller does (usually at an instruction boundary). If there is no interrupt, the next state control selects control store addresses from the output of the control store, the branch control unit, or the instruction decoder.

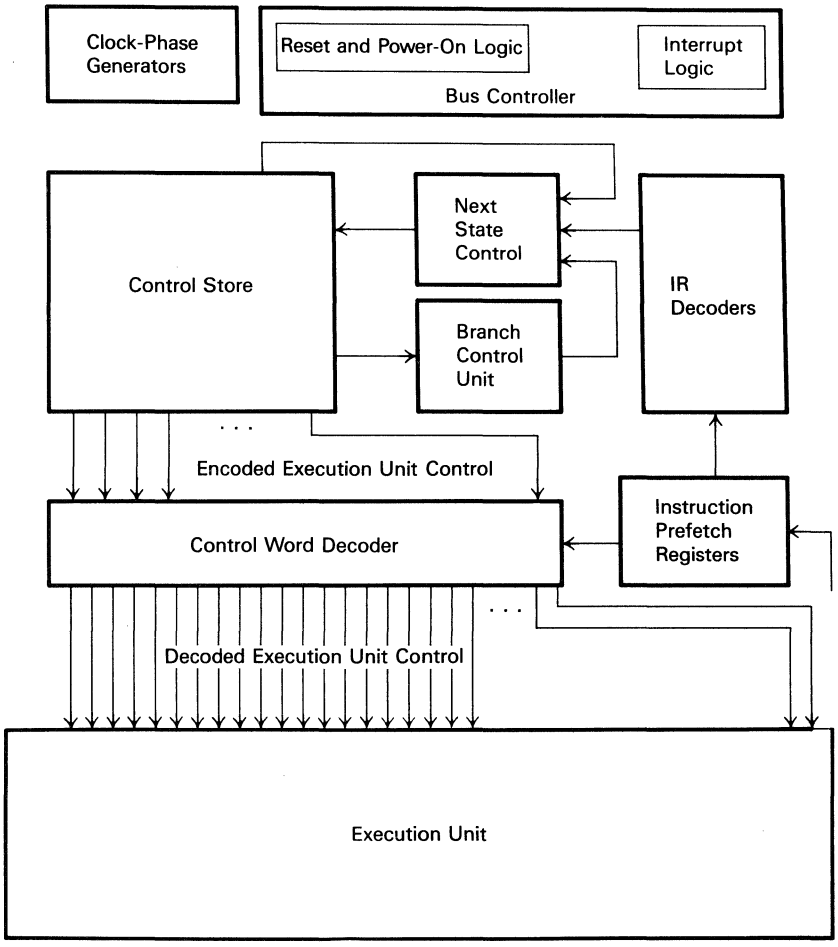


Figure 2.6 Microprocessor block diagram

The branch control unit provides the means for decision-making in the microcode. Both Micro/370 and the MC68000 provide four-way branches using a partial address from the control store and altering some of the control store next address bits based on conditions in the execution unit.

The control store holds the control words. Part of each control word is decoded to control the execution unit elements, and part helps run the state sequencer (by saying where to get the next control store address or even by providing it).

The control word decoder translates the (compact) control word into the exact lines needed to control each execution unit element. The control word decoder also mixes the information from the control store with information from the instruction register and with timing information. In the MC68000, the part of the control word that helps run the execution unit is 66 bits wide. It is decoded into about 180 bits to run the execution unit control points. In Micro/370, the part of the control word that helps run the execution unit is 71 bits wide. It is decoded into about 300 bits to run the execution unit control points.

An execution unit control point is a single control line leading to a macro in the execution unit. For example, a single control line might gate the value of a single register onto a bus. The load signal for a register would be another single control line. The op code control for the ALU might be four control lines. Each of these lines becomes a control point entering the execution unit from the processor controller.

The instruction prefetch registers allow the microprocessor to overlap the execution of the current instruction with decode of the next instruction and with the prefetch of the halfword after the next instruction. There are three registers. One holds the currently executing instruction and is used by the control word decoder. Another holds the next instruction and drives the instruction decoder. The last register receives the halfword following the next instruction, when the halfword arrives from the external bus.

The bus controller runs the electrical protocol to communicate with the outside world. The bus controller detects and synchronizes external interrupts, runs memory access cycles, and arbitrates control of the external bus.

This is a preview of the explanation of how a microprocessor works, presented in chapter 5.

C H A P T E R

3

Hardware Flowcharts

The Flowchart Method is the procedure and notation I use to design the CPU of a computer. The method works for general-purpose and for special-purpose CPUs.

A CPU has a "controller" and an "execution unit." The execution unit is a collection of fast but latent capabilities (registers, ALUs, shifters, and data paths). The controller controls the execution unit by telling the execution unit what to do when. The controller determines the CPU's "personality."

Designs often begin with an appeal: "We need a CPU that's twice as good as any rival's." Computer architects turn the appeal into an English description of the machine (in IBM's System/370, this is the *Principles of Operation* manual, form no. GA22-7000). Engineers implement from the English description, using logic design and circuit design methods. We have lots of books to help us with logic design and circuit design, but nobody says how to transform the English description into the kind of formal description circuit designers need.

It's much like a mathematical word problem. The hard part is getting the equations from the written description of the

problem. Once you have the equations, you can apply documented methods to find the solution. The English description of a chip is like a book-length mathematical word problem. Hardware flowcharts are a bridge between English and the logic designer; they are a compact formal description of what the CPU does.

The method I describe was used to design the controller for the Motorola MC68000 and IBM Micro/370 microprocessors. The flowchart method is both procedure and notation. The designer follows the procedure to express the design in the particular form I call flowcharts. Unlike most procedures, this one does not start out by presuming a block diagram for the controller. (Doing this imposes a structure on the English specification; the problem is to *find* an efficient structure.) The block diagram is one of the procedure's *outputs*.

Flowcharts show the design as the flow of simple actions. An example is RX→A→ALU, which means "put the contents of register RX on the A bus to the ALU." (That also exemplifies the notation; it doesn't get more complicated than that.) One of these statements is called a task; states can be one (really zero) or more tasks. I depict the flow of states by boxes (one for each state); I draw these in a specific format, and it is important that you draw the states precisely the way I say. With the flowchart method, you see major flow (a complicated microprocessor can fit on six 8½-by-11-inch pages) without losing important detail. RX→A→ALU is uncluttered by the usual hardware details that hide significant controller structure issues. The hardware is debugged using the flowcharts; they are the authoritative reference for the design.

The procedure is carried out with a particular technology in mind (flavors of bipolar, nMOS, CMOS). Decisions in the procedure are based on the capabilities of the particular technology. The procedure does not depend on the implementation method. This means that the same flowcharts are used to implement the chip with combinational logic, PLAs, or microcode. In chapter 4, I show how to implement a simple microprocessor using flowcharts.

I tell how to flowchart hardware using just pencil and paper. I describe flowcharting using such simple tools because:

1. The method is useful whether the designer has just a desk and wastebasket or several million dollars' worth of computers and fancy equipment.

2. Design automation should be subservient to the design method. It should support the design procedure, not *be* the design procedure. (Often, engineers' methods are solely the result of available design automation tools; I think that's bad.)

Prerequisites

Flowcharts tell how to get from the architecture to the implementation. They link the programmer's (external) model and the hardware (internal) implementation. Flowcharts specify exactly how commands from the instruction set are carried out using execution unit hardware. You must have the instruction set summary and an execution unit specification before you begin flowcharting.

Instruction Set Summary

The instruction set summary is published as a necessary part of the user's manual. (See, for example, the *MC68000 User's Manual* for the Motorola MC68000 or the *iAPX Book* for the Intel 8088.) The instruction set summary describes:

1. Instruction formats
2. Operations (ADD, AND, SUB, and so on)
3. Addressing modes (Base Plus Displacement, Register Indirect, Indexed, and so on)
4. Registers (as seen by the programmer)

Execution Unit

A microprocessor's execution unit (or data path) details are not usually published for several reasons: Users do not want to know, users should not know, or manufacturers want competitive advantages kept secret. You need a block diagram of the execution unit that shows the following:

1. Programmer's register set
2. Additional registers (such as the instruction register, program counter, and temporary registers)
3. ALU and any special function units (such as a shifter)
4. Internal data paths
5. Rules of operation

All this information (except maybe some rules of operation) should be in the execution unit block diagram. The rules of operation tell what can and cannot be done with the execution unit

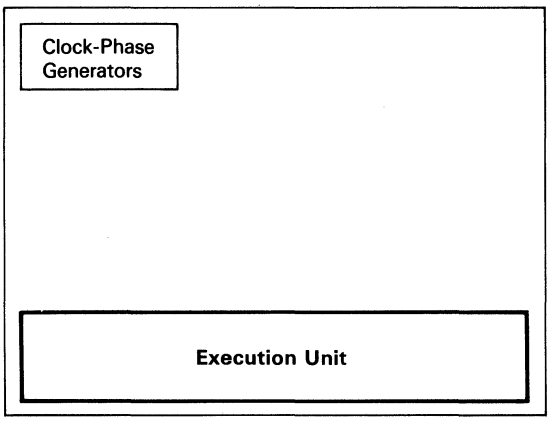
pieces (registers, buses, arithmetic units, and so on). The rules of operation also tell clock phases, timing, and electrical load constraints for the pieces. These rules are imposed by circuit design limits.

If you are responsible for the flowcharts, you should do the execution unit first. To design the execution unit, I recommend doing trial flowcharts for ten frequently used instructions to determine an initial execution unit structure. I think a simple bus-oriented structure is best, so I start with that. In a current (1987) very large scale integration (VLSI) implementation, some limits on your interconnect scheme will come from the circuit designers. For example, having no more than three buses allows bus wiring to pass right over the registers and arithmetic units without using extra chip area.

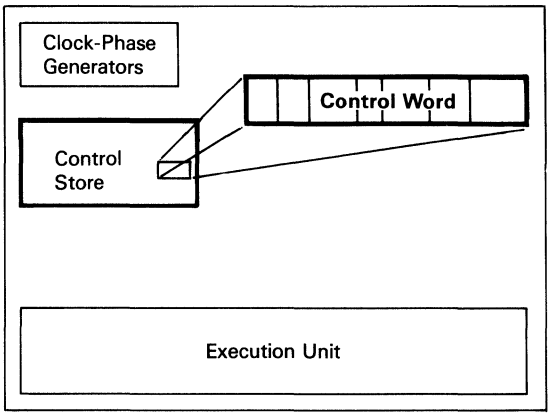
The execution unit will evolve. I proposed the initial execution unit for Micro/370 in January 1981. It went through about twenty-three major revisions before I completed the flowcharts. These changes are expected—and are supposed to happen. For example, in writing flowcharts for the instructions, you find an instruction you cannot implement efficiently. You can't do a Booth's algorithm multiply efficiently because you can't "see" the low-order bits in the multiplier. Since the multiplier normally resides in the shifter, you just wire the low-order shifter bits to the branch control unit. Perhaps you need a special direct path from the ALU to the Data Temporary register (DT). You can just move the DT next to the ALU and wire the direct path. If you need something, add it. The circuit designers will tell you when you're not being reasonable.

Illustrated Flowchart Method Overview

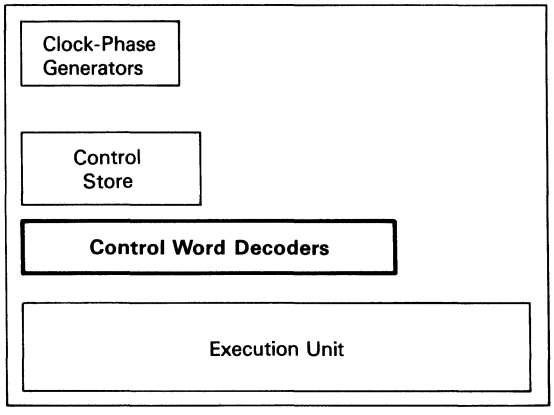
Figure 3.1 shows the development of the implementation using the flowchart method. To avoid confusing details, I illustrate the method with a simple microprocessor, called MIN. Figure 3.2 shows the instruction format and register set; figure 3.3 shows part of the instruction set summary. This subset is adequate to demonstrate flowchart construction. Figure 3.4 shows a sufficiently detailed block diagram of the execution unit. It also includes some rules of operation; others will be added as I progress.



- The architecture specification is the only input.
- Begin with a guess for the execution unit.
- Do flowcharts for the instructions.
- This modifies and refines the execution unit and develops the control store and control strategy.
- The final execution unit is derived output.

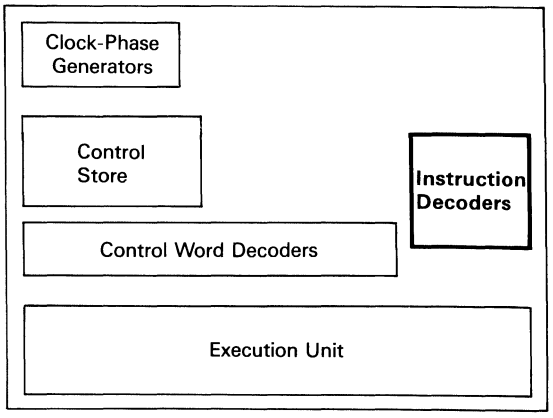


- Once the flowcharts are fairly complete, derive the control word format using the flowchart states.
- When the flowcharts are complete, so is the execution unit.
- Control word format is derived output.

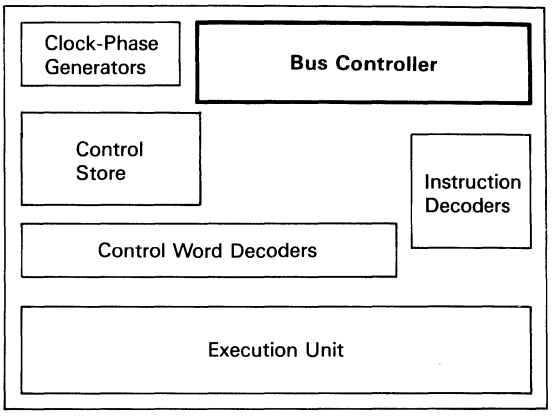


- After defining the control word format, you assign bit patterns to the control fields in a way that minimizes control word decoders between the control store and the execution unit.

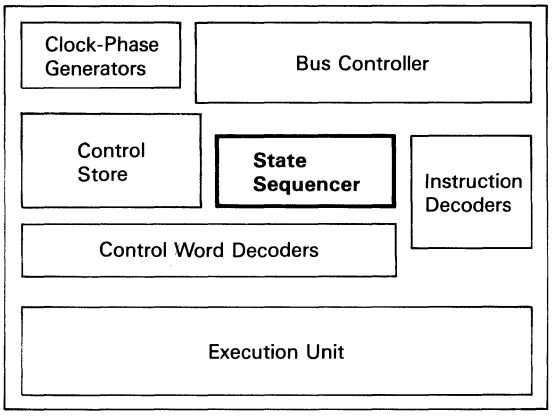
Figure 3.1 Development of implementation using the flowchart method



- Instruction decoders are defined by the flowcharts and the architecture specification.



- Completed flowcharts, control word format, and the initial bus specification define the bus controller.



- Last is the logic of the state sequencer, the part of the chip that says what to do next. ("Where's the next control word?")
- Once everything around it is defined, you build exactly what you need! (The state sequencer is derived output.)

Figure 3.1 (continued)

Instruction Format

First Word

OP	RX	Mode	RY
Operation code	First operand register	Second operand address mode	Second operand register

Second Word

Displacement

Optional, depending on second operand address mode

Programmer's Register Set

R0
R1
R2
. . .
Rn

Figure 3.2 MIN instruction format and register set

Some Operations

ADD
AND
BZ — Branch if zero bit is set, register indirect only
LOAD — Second operand is source, and RX is destination
POP — Postincrement with register indirect only
PUSH — Predecrement with register indirect only
STORE
SUB
TEST

Second Operand Address

Mode	RY
Second operand address mode	Second operand register

Address Modes

AB-Base (RY) plus displacement (second instruction word) is an operand address.

AI-Register indirect. RY holds an operand address.

AR-Register direct. The result is stored in RY. For two operand instructions, RY also is an operand source.

Figure 3.3 MIN instruction set summary

Figures 3.2, 3.3, and 3.4 do not include the usual details about word length, instruction length, address length, bus width, ALU size, and register size. Although you know this information, it doesn't change the sequence of operations for the execution unit. The sequence of operations depends on relative values of these parameters and not on their absolute values. You implement the design from the flowcharts with a particular word length, instruction length, address length, and so on. Don't clutter your flowcharts (or your notation) with details you don't need.

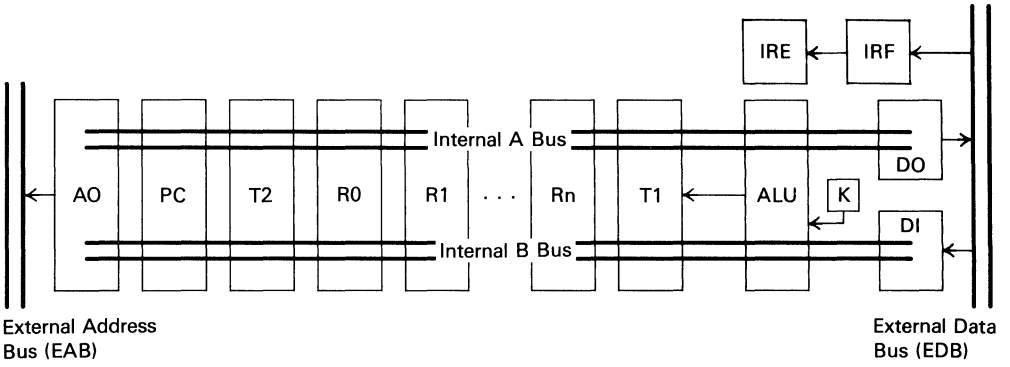
Flowchart Objectives

Now you have ample information to construct flowcharts, but you face some difficult questions: What are the design objectives? Which objective is most important? Next? Least?

Here are some reasonable design objectives:

- Limit controller size to some fraction of a single chip. Since profit goes up as die size goes down, there will be pressure to make the controller smaller even when it fits.
- Make the CPU as fast as possible (certainly faster than its contemporaries).
- Complete the project early to give the product an early start in the market.
- Make the flowcharts easy to translate into hardware.

This illustrates the value of a good project manager: He or she ranks the objectives.



ALU	Arithmetic and Logic Unit	DO	Data Out buffer	K	Constant generator
AO	Address Out buffer	IRF	Instruction Register for Fetch	PC	Program Counter
DI	Data Input register	IRE	Instruction Register for Execution	R0-Rn	Programmer's registers
				T1, T2	Temporary registers

Example Rules of Operation

1. A transfer from source to bus to destination takes one state time.
2. A source can drive up to three destination loads.
3. Inputs to the ALU are from the A (internal) bus and either K (values 0, +1, −1) or the B (internal) bus.
4. When the ALU is a destination, T1 is automatically loaded from the ALU output.
5. A transfer to AO activates the on-chip external bus controller.

Figure 3.4 MIN execution unit block diagram

I have chosen an execution unit with a simple two-bus structure for the MIN CPU example. You talk to the circuit designers about what structures are reasonable for the technology. You arrive at the proposed execution unit by doing some trial flowcharts. Figure 3.4 is the proposed execution unit. Here are some rules of operation:

1. A transfer from source to bus to destination takes one state time. (It takes one flowchart state to execute the task RX→A→ALU, for example.)
2. A source can drive up to three destination loads. (For example, the task T1→B→ALU,AO,PC has three destination loads: ALU, AO, and PC.) The circuit designer will tell you how many destination loads each source can drive.
3. Inputs to the ALU are from the A (internal) bus and either K (values 0, +1, −1) or the B (internal) bus. One side of the ALU has one input source (A) and the other has two input sources (K and B).
4. When the ALU is a destination, T1 is automatically loaded from the ALU output at the end of the state time.
5. A transfer to the AO buffer activates the on-chip external bus controller. This bus controller postpones the next state until the external transfer is complete.

Picking the initial execution unit requires some knowledge of implementation cost for the technology you use. The circuit designers should help you with this. It is much better to start with too little than too much. (It's easier to add things than to figure out whether you can throw them away.) Start with a simple execution unit and add resources as you need them. If you begin with an extravagant guess, you may build something fancier (bigger and slower) than you need. The flowchart method can help you identify features that improve performance. It will not tell you what you don't need. And it will not tell you when you have an overkill (too much hardware for the problem you are solving). The circuit designers should warn you when you are asking for more than they can do. If they trust you, however, they will try to build what you want—even if it is too much. It all comes down to this: You are an engineer (the logic designer). You have to use restraint, common sense, and judgment. I can't find a procedural substitute for *you*. I can only tell you what helps me.

Making a Flowchart

When it is time to begin the flowcharts, you will be plagued with all sorts of questions. How do I begin? What do I write? How do I write it? I suggest methods that work for me. Use a register transfer notation to describe the operations of the execution unit. Each statement in this notation is called a task in the flowcharts. Each state comprises one (really zero) or more tasks. Use rectangles for states. (In a microcoded implementation, each state becomes a control word.) A control word sequence is a succession of states. Work on large sheets of high-quality graph paper (preferably 17-by-22-inch vellum with ten lines per inch). Large sheets make it simpler to see and to plan large segments of the control flow, and high-quality paper lasts through many changes.

It can take years to complete the flowcharts for a complicated CPU. (It took me a year to complete flowcharts for the Motorola MC68000 and about two and a half years to complete flowcharts for Micro/370.) To avoid copying several generations of flowcharts, observe these rules:

- Work in pencil. (Use a .5mm Pentel with F lead.)
- Work on the back of the vellum so you won't erase the grid.
- Use an erasing shield and an electric eraser.
- Always use a cover sheet to prevent smearing.
- Plan changes on scratch paper and transcribe them to the vellum.
- Always use reproductions for work and reference. (I reduce the copies to 8½ by 11 inches for easier use.)
- Accumulate changes (in red ink) on a reproduction.
- Do trial level 2 flowcharts (level 2 flowcharts are explained later) on 8½-by-11-inch scratch paper with 1½-inch-high, 2-inch-wide penciled-in rectangles as guides. (I load the copier with junk memos and copy the grid on the back.)

Figure 3.5 shows flowchart sequences for the register-to-register ADD instruction, the register-to-memory ADD instruction using the MIN execution unit (figure 3.4), and a simple register transfer notation. Each box is a state. Each line entry in a state is a task. Tasks are expressed in the register transfer notation; the notation has a source-bus-destination format. Alphabetize tasks

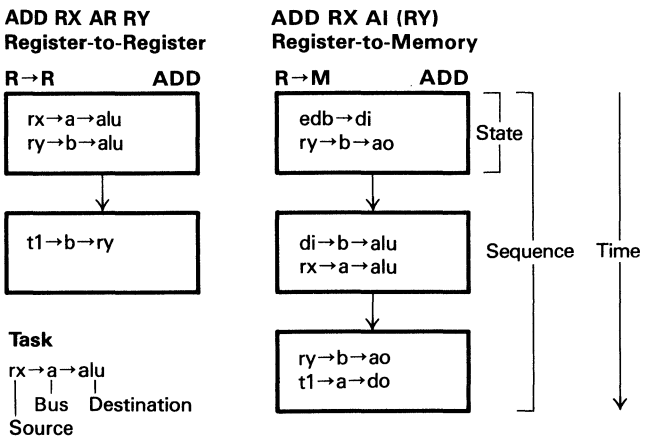


Figure 3.5 Execution of register-to-register ADD and register-to-memory ADD instructions (partial description—operation tasks only)

in each state by source (if there are multiple destinations on a single line, alphabetize them, too); you will use this to compact the flowcharts later on.

In figure 3.5, time advances from the top of the page to the bottom of the page, except within a state. Within a state, tasks appear to be concurrent but are governed by rules-of-operation timing. In a microcoded implementation, each state is one microcycle (and may have phases such as source, transfer, destination, and precharge).

In the register-to-register example (the left flowchart in figure 3.5), I transfer both operands to the ALU in the first state. The output of the ALU is saved in T1 any time there is an ALU operation. In the second state, the result is sent from T1 to RY. Look at the register-to-memory ADD example (the right flowchart in figure 3.5). The first state fetches the memory operand; the second state adds the operands; the third state sends the result to memory.

Something doesn't look right. In the first state, DI is loaded from the external data bus (EDB), but RY is sent to AO after this happens. Wrong! I consider these tasks concurrent (with some implicit timing). They are in alphabetical order. Sending RY to AO initiates the external bus activity that results in the DI transfer from EDB. The tasks are listed in the same state because as far as the state sequencer is concerned, they happen at the same time.

In a microcoded controller, the control word specifies the tasks in a single state. The tasks are commands to the external bus controller, the execution unit, and the state sequencer. Whatever timing is added later, the commands all come out of the control store at the same time. In the case of a read, the transfer to AO initiates the external bus cycle. If the external bus is synchronous, then DI must be valid by the end of the state time. If the bus is asynchronous, the state sequencer "hangs" in the current state until the transfer to DI is completed. In the case of a write, the address and data are transferred to the external address bus (EAB) and EDB, respectively. The state sequencer "hangs" until the external bus controller signals the state sequencer that the external transfer is completed.

There is no explicit notation for transfer from AO to EAB or from the Data Output register (DO) to EDB (or for memory to EDB). I have elected to let them be implied by the context. I view AO and DO as amplifiers (not registers). Because AO is not a register, it does not remember the address between state 1 and state 3 of the register-to-memory sequence (the right flowchart in figure 3.5). Transfers from EDB to the execution unit are not implicit because they can be to the instruction register for fetch (IRF), DI, or both.

Notation

Keep the register transfer notation simple. It must capture the essence of what the CPU is doing without all the details. You may think this is a simple notation invented for just this one case. Well, that's somewhat true. I modify the notation to fit the problem. I want the notation to be a simple, natural, readable way to express what the CPU is doing. That is why the notation is not formally defined. In a formal notation, constructs might prevent natural expression of tasks and hinder the design.

Flowcharts are graphic notations that depict the CPU in two ways:

1. Flowcharts visually emphasize changes in sequence and concurrency for whatever the controller is doing. You see branching and merging in the flow of control. You see how the address calculation sequences and operation sequences are shared. You see all the instructions sharing one common set of address calculation sequences. You see ten instructions sharing the standard dual operand execution sequence

(as the MC68000 register-to-register operand execution sequence does, for example). You see which instructions have an execution sequence all to themselves (multiply or divide in Micro/370, for example).

2. Flowcharts visually communicate the relationship of sequence to concurrency for whatever the controller is doing. You see exactly what is concurrent (tasks) and what is sequential (states), and you see how they are related.

Flowcharts show sequential state flows made up of concurrent tasks. Each task is a sequential source-bus-destination flow. Flowcharts are a flow-intensive notation showing you the concurrent and sequential nature of operations.

Execution Speed

The flowchart sequences in figure 3.5 are incomplete. They do not include the instruction fetch and the PC increment. The PC increment and instruction fetch could be added to the beginning or end of both sequences (with different consequences). Which leads to the fastest controller? Just what is the fastest controller? How about this definition: The most efficient controller executes a given instruction with the least number of states.

"That's kind of a truism. Give me something I can use—that tells me what to do." You are designing something (a microprocessor) that will be part of a larger system (a board, a personal computer, an instrument). What limits system performance? Is it always your part? Sometimes your part? If your part is the system bottleneck, you did not design it very well. If your part is never the bottleneck, perhaps you spent too much on hardware. The best engineering design achieves the effect of infinite resources (never the bottleneck) at minimum cost. Microprocessor design is a good example. I believe that useful external bus activity in every state is evidence of sufficient controller efficiency. Therefore, I use the following definition for controller efficiency: The controller is efficient if execution never delays external bus cycles. (If some other part of the system is the bottleneck, the controller design is good enough.)

Measuring the microprocessor performance at the pads will not reveal whether you implemented a Cray supercomputer or a controller barely sufficient to make external bus transactions the bottleneck. This is not a measure of bus efficiency or system efficiency; it is a measure of how well you do the controller design.

I can't give a useful general definition for the fastest controller because it depends on what the controller does. I have given a definition that works for a microprocessor, but an applications engineer would not use this definition because he would not want the external bus tied up by the CPU all the time.

Figure 3.6 improves the examples in figure 3.5 with the PC increment and instruction fetch. I removed the lines connecting boxes because they are unnecessary and doing so saves space. The more states you fit on a page, the more of the design you take in at a glance. (I still use lines to show the next states of sequences with internal branches.) To make a quick measure of efficiency possible, I put a shaded box in the upper right-hand corner of states with external bus activity. Assuming states of equal duration, the overall efficiency of the execution unit is 20 percent for the register-to-register instruction and 50 percent for the register-to-memory instruction. Our competitors will be pleased. What can I do about it? In some states of each flowchart sequence, the major internal buses (A and B) are not both occupied. That's not good. It should be possible to merge tasks for greater efficiency. We must find a way to squeeze more performance out of the execution unit.

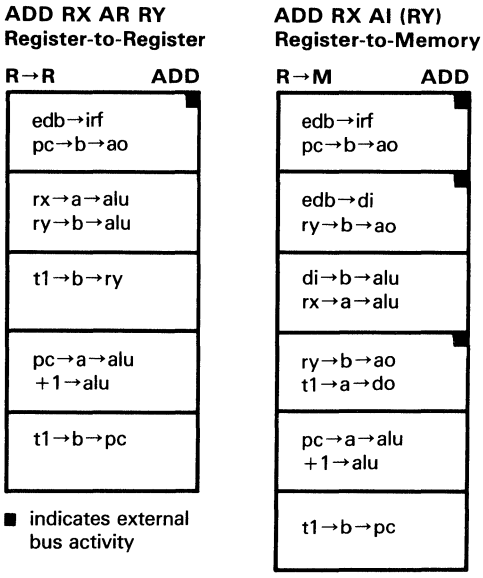


Figure 3.6 Revised execution of ADD instruction examples

Level 1 Flowcharts

Separate an instruction's execution into operation tasks and housekeeping tasks and treat each differently. Operation tasks are transfers required to perform the instruction. These tasks (such as accessing operands, storing results, and moving data to and from the ALU) must occur in a specific order and may be unique to a particular instruction. Figure 3.5 shows the operation tasks for two types of ADD instructions. Housekeeping tasks, such as PC increment and next instrument fetch, are common to all instructions. You have some leeway in deciding when these tasks are accomplished. The tasks are essentially independent for all instructions, so you should treat them separately (initially). Separate kinds of tasks so you can optimize the execution of the operation tasks.

Figure 3.7 shows the flowcharts in a format designed to aid later merging of operation tasks and housekeeping tasks for maximum execution efficiency. This is the level 1 flowchart format. For each instruction, operation tasks are in the left sequence, and housekeeping tasks are in the right sequence. Do level 1 flowcharts for most of the instructions and then begin level 2 flowcharts. You do not have to do level 1 flowcharts for all instructions. If you have instructions for which housekeeping tasks are an insignificant portion of the execution time, it is a waste of time to do level 1 flowcharts. For example, the System/370 MVCL (Move Character Long) instruction may take several thousand states to execute and has only a couple of states of housekeeping tasks. It isn't worth doing twice (once in level 1 flowcharts and once in level 2 flowcharts). Level 1 flowcharts find the best execution sequence for the operation tasks and identify the housekeeping tasks. Level 2 flowcharts merge the housekeeping tasks with the operation tasks. The direction of the merge is into the operation tasks. (You want to make the housekeeping tasks "disappear" into the operation task sequence.) The state order of each column must be preserved in the final sequence (called the execution sequence), but housekeeping tasks can be merged with operation tasks wherever reasonable. (We shall see consequences of this merging later.)

You would achieve the most efficient execution (for this execution unit) if you merged the housekeeping tasks with the operation tasks without increasing the number of states in the operation task sequence. Usually, it is adequate to have the number of

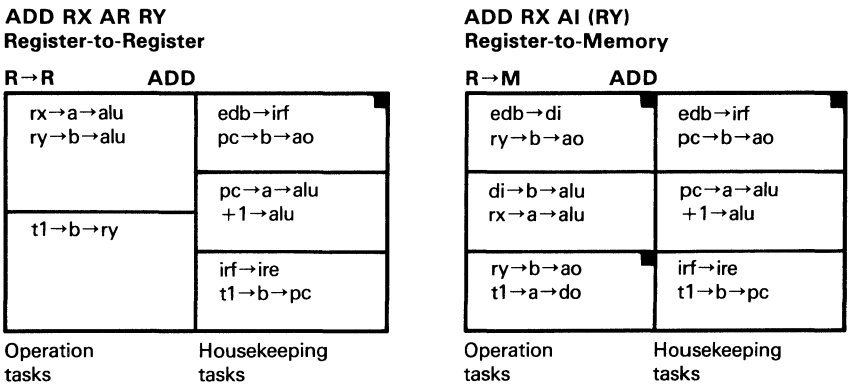


Figure 3.7 Level 1 flowcharts for two types of ADD instruction

states in the final execution sequence be significantly less than the total states in housekeeping task and operation task sequences. Since the microprocessor allows only one external bus access per state, you have done enough when you merge housekeeping tasks into an operation task sequence in a way that produces useful external bus activity (including housekeeping accesses) in every state.

Increased speed may not be the only objective of the merge. You also should merge the tasks to create as many identical states (across instruction types) as possible. I assume that a controller with fewer unique states is smaller.

Be careful merging housekeeping tasks into operation task sequences. You would not want to increment the PC before you computed a PC-relative branch address. Merging housekeeping tasks into operation task sequences is challenging and fun because it requires skill and care. You may reorder tasks, change the execution unit, and try dozens of combinations and sequences to get the most efficient execution sequence. This is design. You are working to find the best execution unit for the instruction set and the best controller for the execution unit. If you like puzzles, it won't even seem like work. This is how you are creating the controller. I will show (later) how assumptions you make in the flowcharts translate to hardware in the controller. You get the controller you need, which is better than choosing a controller and trying to make it do what you need.

I added one more thing in figure 3.7. IRE is the instruction register for execution (see figure 3.4). It allows a rudimentary prefetch. IRE holds the current instruction and drives the register selection decoders (for RX and RY). IRE is loaded at the beginning of a state, and decoding will be stable within one state time. It must not be changed until after the last RX or RY reference in the flowchart sequence for the current instruction. Each instruction (sequence of operation and housekeeping tasks) is associated with a particular register pair (RX, RY) established by IRE. The instruction register for fetch (IRF) can be used to hold the next instruction until the current instruction is done. It can be loaded anytime during the current instruction—this is the simple prefetch. More accurately, IRF gets the word following the current instruction. (It may not be the next instruction if the current instruction is a branch or a two-word instruction.)

Level 2 Flowcharts

Figure 3.8 shows the housekeeping tasks merged with the operation tasks to form what I call level 2 flowcharts. The efficiency of the register-to-register sequence is 33 percent, and the efficiency of the register-to-memory sequence is 75 percent. (You could do better with a more complicated execution unit and controller.) Register T2 saves the operand address in the register-to-

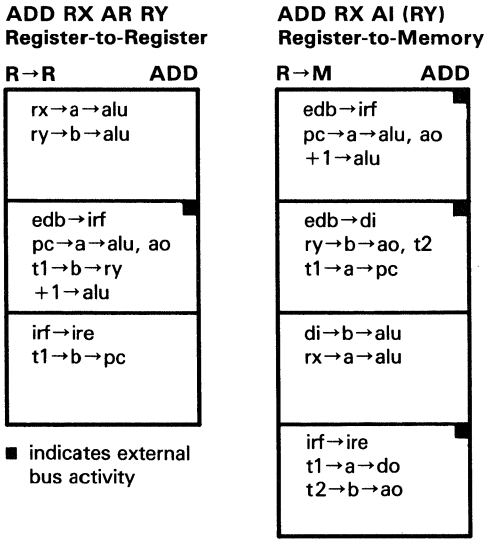


Figure 3.8 Experimental reduction of the level 1 flowcharts

memory ADD example. Because T2 contains the memory address (for the second operand) and the static decoders (which are driven by the IRE) are available, there are no more RX or RY references. The last state can change IRE and store the result.

Feedback on Execution Unit Design

Do a level 2 flowchart of the fastest instruction. This will point to inadequacies in the execution unit design. In general, you will discover inefficiencies in the structure of the execution unit as you merge the housekeeping tasks with the operation tasks. In the register-to-register ADD example, if the AO buffer had not been accessible from the A bus (see figure 3.4), I would not have been able to do the instruction in fewer than four states. Less than full use of the A and B buses in the resulting sequence would signal the need to improve the execution unit.

Figure 3.9 shows a register-to-register ADD sequence for an execution unit with no path from the A bus to the AO buffer. Beware! The increased complexity of the execution unit can increase the number of unique states and result in a larger controller. Increasing the complexity of the execution unit implies more execution unit hardware, too. Only after carefully studying the flowcharts and the execution unit would I suggest execution unit changes to improve the efficiency of the overall design.

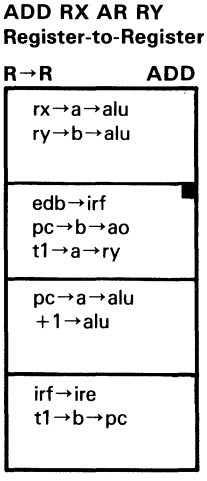


Figure 3.9 ADD sequence for an execution unit in which AO is connected only to the B (internal) bus

The flowchart method helps design hardware. I believe that it is a good, workable method. It is not a rote recipe for good design; you still have to know something about what you are doing. For example, if you start with a fancy execution unit, there is nothing that tells you to throw away expensive hardware. A little arrow drawn on your execution unit may imply a 32-bit data path (lines, space, power) with pass transistors and control signals. Be sure you need it. You should know the cost of what you ask for. Start with a simple execution unit and add what you need. The circuit designers should tell you when you ask for too much.

Feedback on Controller Design

Use the format in figure 3.7 to create level 1 flowcharts for the entire instruction set. How many sequences is that? The upper bound is 2^w if w is the instruction length in bits. That is too many, however, because I write only one sequence for each instruction—independently of which registers are specified. (This is an advantage of static decoders for the register fields.) I need only decode the op code and the mode bits in the effective address field (see figure 3.3). Suppose the simple MIN CPU has k operations (ADD, AND, OR, SUB, and so on) and a address modes (Register Indirect, Base Plus Displacement, Indexed, and so on). If any address mode is valid for any operation, I would need $k*a$ instruction sequences.

Clearly, this number can be large. For example, the Motorola MC68000 has about 14 address modes and more than 50 instruction types. If an average instruction has 8 states, then I must implement more than 5,600 (50*14*8) states in the controller. Such a chip would make a good office partition.

Note that I have segmented the flowchart sequences for executing instructions. The sequence of flowchart states for an address mode calculation (which may or may not include operand fetch) is called an address mode sequence. The sequence of flowchart states that completes instruction execution (once the address mode sequence is finished) is called an execution sequence. The combination of an address mode sequence and an execution sequence forms a control word sequence. If an instruction (such as a register-to-register ADD) does not need an address mode sequence, then the execution sequence and the control word sequence are the same. I will use this terminology throughout the book.

If most address modes can be used with most operations, why not share address mode sequences? (Address mode sequences calculate the operand address, fetch the operand, and place it in DIN.) ADD Register Indirect and OR Register Indirect, for example, would share a common Register Indirect address mode sequence. Also, the Register Indirect address mode sequence and the Base Plus Displacement address mode sequence could branch to the same execution sequence for the OR (or any other) instruction. The operand will be in DIN; the execution sequence doesn't care how the address was calculated to put it there. If you share the address mode sequences among the execution sequences, you need only $k + a$ sequences, and that is in keeping with the goal to reduce controller size.

This is a good idea, but what will it cost? It's not free. Suppose you enter the execution sequence, jump to an address mode sequence (subroutine), then return to the execution sequence to complete execution of the instruction. Such a subroutine call costs time (branching to and returning from the address mode sequence), but it lets the controller be much smaller (since the address mode sequences are shared by the execution sequences). The size and speed goals conflict, so a trade-off is in the offing.

How important is the time lost in these subroutine calls? To find out, have the instruction set designer rank the instructions in order of importance. The designer could base the ranking on static or dynamic frequencies of occurrence. However the designer does it, if she designed the instruction set, she must take the stand on what is important. A ranking for the sample MIN instructions is shown in figure 3.10.

Sharing sequences reduces controller size. From the ranking you see that slow subroutine calls are costly because at least three

LOAD	**Most important**
BZ, . . . (other branches)	
STORE	
ADD, AND, SUB, . . .	
TEST	
PUSH	
POP	**Least important**

Figure 3.10 Ranking of MIN instructions

of the four most important instruction types can use any address mode (hence, would have to branch to and return from an address mode sequence). You will not use subroutine calls. You assume that address mode sequences can be shared by initially entering the address mode sequence and branching directly to the appropriate execution sequence. One way to do this in a microcoded controller is to have the instruction decoder provide more than one control store address—one for the address mode sequence and one for the execution sequence.

Flowcharting has led us to a functional requirement for the controller. (The instruction decoder is to provide more than a single output.) This shows how controller requirements come from the procedure. You have not, however, constrained the implementation of the controller to be combinational or microcoded; that choice lies in the future. You do not even have a block diagram of a controller, and you do not want one yet because you want the procedure to give you the requirements for the controller independently of what you think a controller should look like. The flowchart method finds requirements for the controller that best fits what the CPU wants to do (the architecture specification).

Doing Level 1 Flowcharts

The level 1 flowcharts for a subset of MIN instructions are shown in figure 3.11 (pages 37–38). In a real CPU, the flowcharts have many more address mode and execution sequences. Note the following things in figure 3.11:

1. At the beginning of instruction execution, IRE is assumed to contain the current instruction. It must be loaded by the previous instruction. Each instruction's control word sequence will, therefore, have to fetch the next instruction and load it into IRE.
2. Instruction execution begins with the address mode sequence (if the instruction has one) and implicitly branches to the appropriate execution sequence for completion. (We will figure out how to build the hardware to support this branching later.)
3. The execution sequences for register-to-register instructions cannot be shared with execution sequences for memory reference instructions. This reduces the savings from sequence sharing.
4. The execution sequences for standard dual operand instructions (ADD, AND, SUB, and so on) are identical except for the

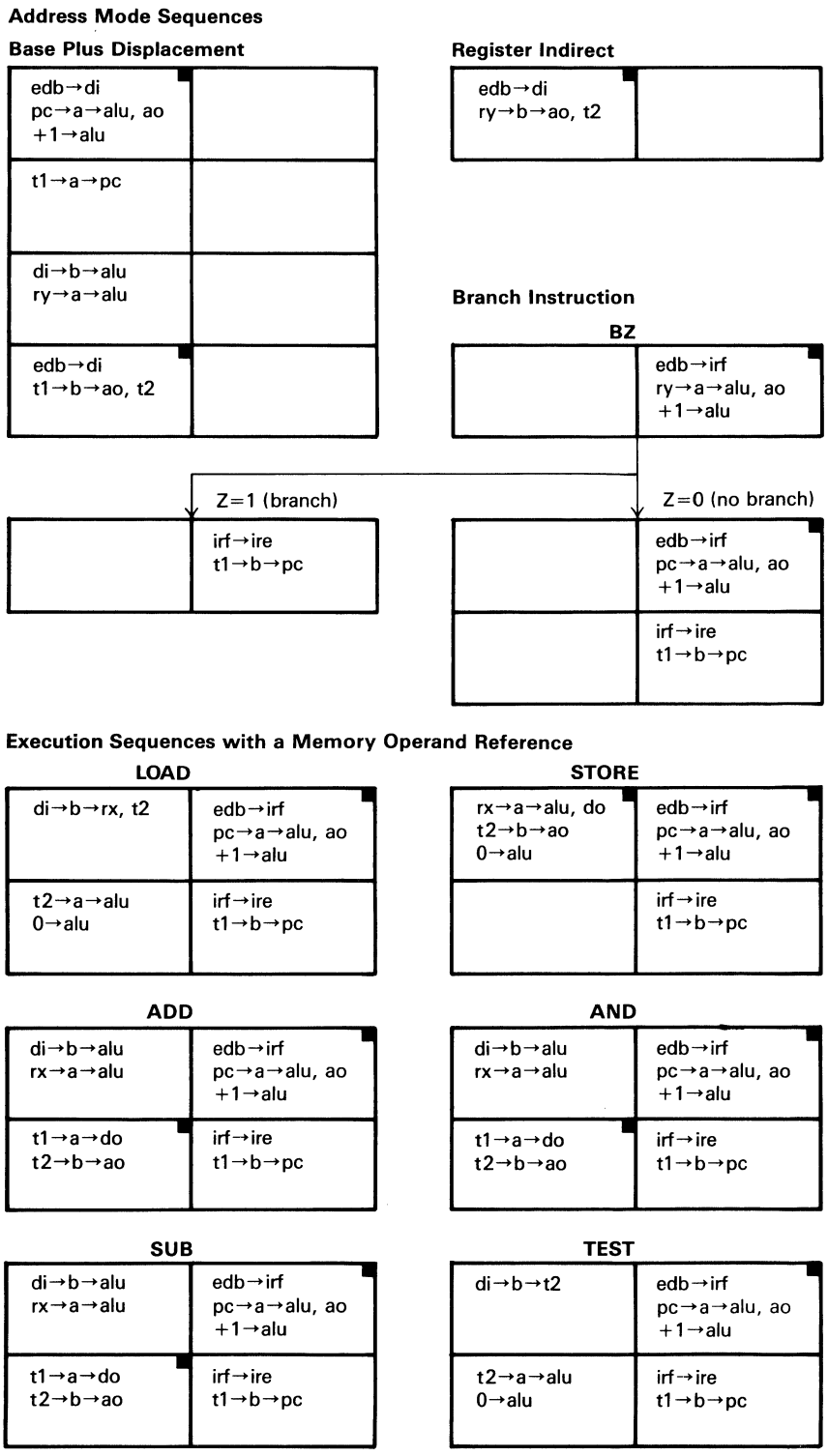


Figure 3.11 Typical level 1 flowcharts for the MIN CPU (continues)

Execution Sequences for Register-to-Register and Special Instructions

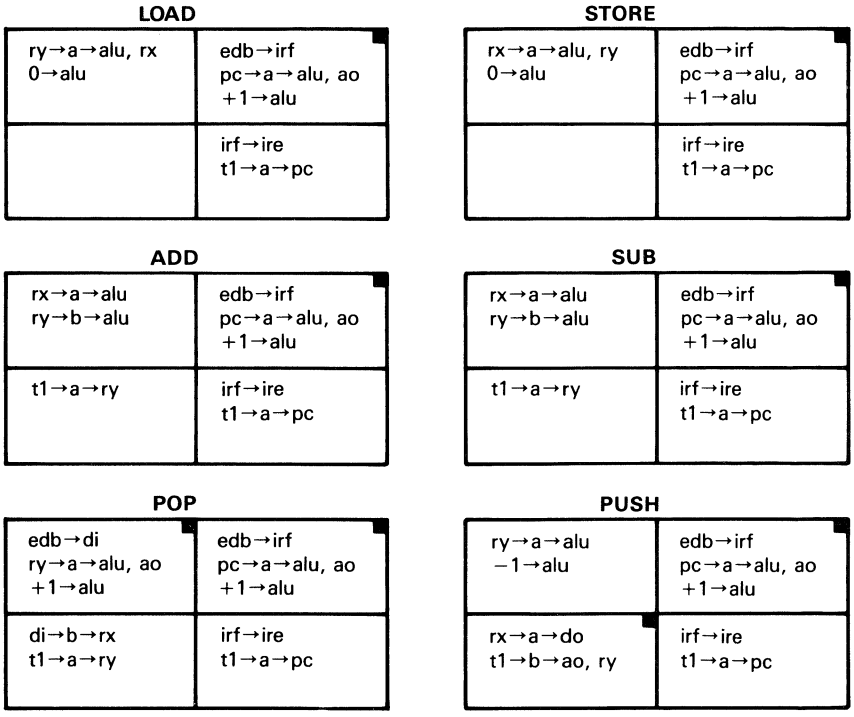


Figure 3.11 (continued)

(implied) ALU function. They can use a common execution sequence if the op code directly specifies the ALU operation (the same way register fields select the registers).

5. Unfortunately, the Store instruction reads the word at the store destination location because it shares the address mode sequences with other instructions. This slows down operation, but I decided to sacrifice speed to make the controller smaller. (There are other reasons besides size why you may not want to do the Store instruction with a read first. Some systems want locations that are read protected. Other systems have memory-mapped peripherals that change states upon a read.)
6. The Branch on Zero (BZ) instruction is a special case. (The Z bit is set to one when a result operand is zero.) Since the condition code (Z) may be set as late as the last state of the previous instruction (in the Test instruction, for example), it may not be available in time to be used at the onset of the

next instruction—in this case, BZ. (Because of the simple prefetch, the instruction decoder is operating concurrently with the execution unit. Information that can change in the execution unit cannot, therefore, be used by the instruction decoder.) As a result, use of the condition code must be deferred at least one state time in the instruction sequence. The branch appears between the first and second states of the task sequence for the branch instruction. Because I need a delay state for the condition code to settle, I must decide how to use the state. The example in figure 3.11 (page 37) shows an anticipated branch prefetch (it is discarded if the branch is not taken). An alternative would be to fetch the next sequential instruction and discard it if the branch is taken. The instruction set designer should be able to tell you which alternative to use.

Note that there are no conditional tasks. If you get to a flowchart state, you always execute all the tasks in the state. There are two types of conditional branches, visible and invisible. One conditional branch is explicitly used in the instruction. Examples are BZ (Branch on Zero), BN (Branch if Negative), and BP (Branch if Positive). Another conditional branch is available to the microcode but not visible to someone using the instruction set. If, for example, you have to implement a Multiply instruction but you do not have a hardware multiply unit, you will do the multiplication with a shift and add or subtract algorithm. You will need conditional branches in the microcode to test the multiplier bits and to detect the end of the algorithm.

Each instruction's flowchart further shapes the design. The functions of the controller eventually will be completely defined by the flowcharts. I have still not constrained the design to be either combinational or microcoded. Once I combine the standard dual operand instructions (ADD, AND, SUB, and so on) into one execution sequence for register-to-register and another for register-to-memory, the level 1 flowcharts are complete. Then I can work with them, merging housekeeping tasks with operation tasks to produce the level 2 flowcharts.

Doing Level 2 Flowcharts

Figure 3.12 shows the housekeeping tasks merged into the operation task sequences for the instructions in figure 3.11. I also integrated the standard dual operand instructions (ADD, AND, SUB, . . .) into a single sequence. (This assumes IRE will select the ALU operation in the same way it selects the registers.)

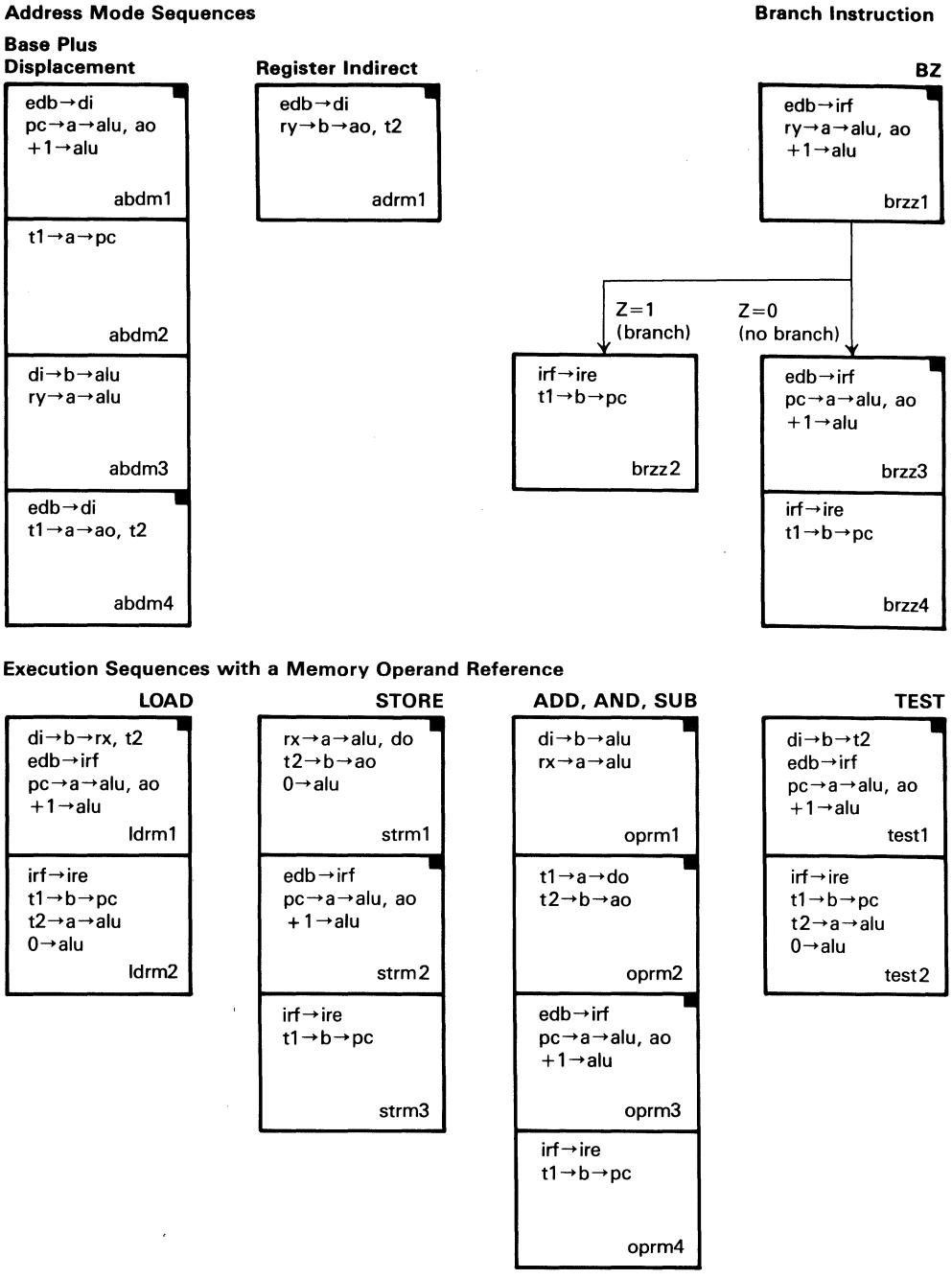


Figure 3.12 Merged level 1 flowcharts for some MIN instructions (continues)

I tried to merge the housekeeping tasks into the operation sequences without increasing the number of states in the operation task sequence. I was not always able to do this (it's a goal, not a requirement). I did reduce the number of states from a potential fifty-four (the number of states in figure 3.11) to an actual thirty-five. (I merged twenty-eight housekeeping states with twenty-six operation states.) Normally, if I cannot reduce the number of states significantly (a matter of judgment), I try to improve the execution unit.

Be careful in merging because operation tasks can use the same resources (such as buses, registers, and arithmetic units) as the housekeeping tasks. Arbitrary interleaving is not possible. For example, if there are PC relative address modes, the PC update (a housekeeping task) must consistently precede (or follow) the address calculation. If a problem during an instruction execution (such as an arithmetic overflow or divide fault) causes an interrupt that stores the old PC value, that value must be consistent

Execution Sequences for Register-to-Register and Special Instructions

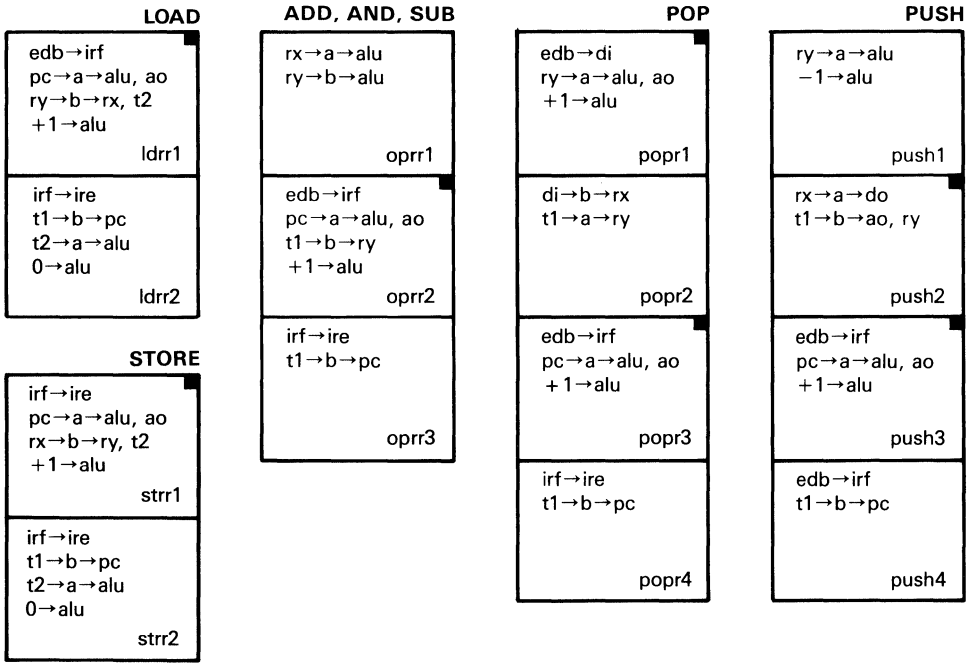
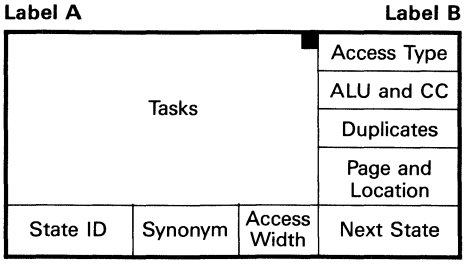


Figure 3.12 (continued)

for all instructions. (The instruction set designer should tell you if you have to store the old PC for interrupts and what the PC should point to.)

Before you can transform the flowchart description to a hardware implementation, you must identify the states. In figure 3.12, I put state identifiers in the lower right-hand corner of each state.

If you add descriptive information, you can ease the transition from flowcharts to hardware, and you make the flowcharts easier to use. But what descriptive information will help? What information do you need? Listed below are some useful kinds of descriptive information for translating flowcharts into hardware. I listed information useful for implementing the MIN controller; a more complicated controller requires more information (for register decoder substitutions or operand sizes, for example). Refer to figures 3.13 and 3.14 (pages 42–45).



Sequence Label A
Operation format or instruction type
Example: **rx op ry→ry**
rx op mem→mem

Sequence Label B
Operation sequence or address mode sequence
Example: **ADD, AND, SUB**
Register Indirect
Indexed

Access Type
DR Data Read
DW Data Write
IR Instruction Read
NA No Access

ALU and CC (ALU function and Condition Code setting)
Example: ADD, AND, or OP condition codes
S Set
N Not set
X Don't care

Duplicates
(Not used in MIN)

Page and Location
(Not used in MIN)

Next State
BC Branch Conditionally
IB Instruction Branch
SB Sequence Branch
State ID Direct transfer

Access Width
(Not used in MIN)

Synonym
(Not used in MIN)

State ID
State Identification

■ indicates external bus activity

Figure 3.13 Format for a level 2 flowchart state

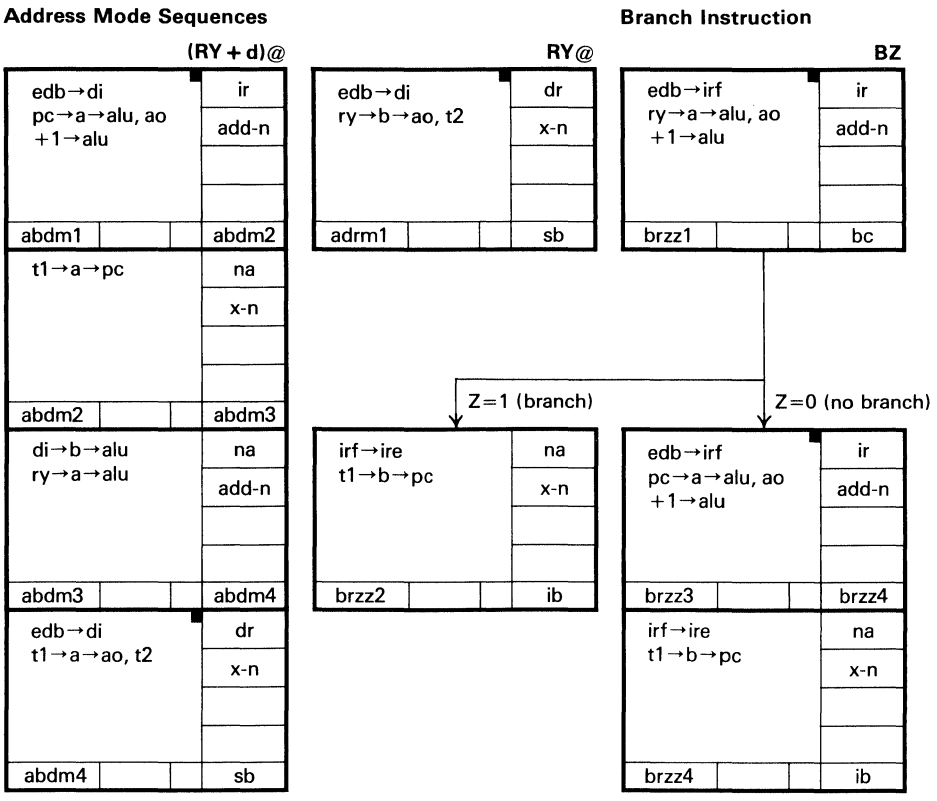


Figure 3.14 Format for final version of level 2 flowcharts (continues)

1. Sequence labels. These labels, associated with each execution sequence or address mode sequence, identify the instructions or address modes using that sequence. They also describe the transfer path. You will use this information later to build the instruction decoders. You can relate an instruction op code bit pattern to the first state ID in the address mode and execution sequence. For example, if IRE contains the bit pattern for register-to-register ADD, you want to begin instruction execution with state oprr1 (see page 45). In a microcoded controller, the instruction decoder will translate the bit pattern of the register-to-register ADD into the control store address of the control word oprr1. In a PLA decoder, the instruction bit patterns will form the AND array, and the control word state ID addresses in the control store will form

Execution Sequences with a Memory Operand Reference

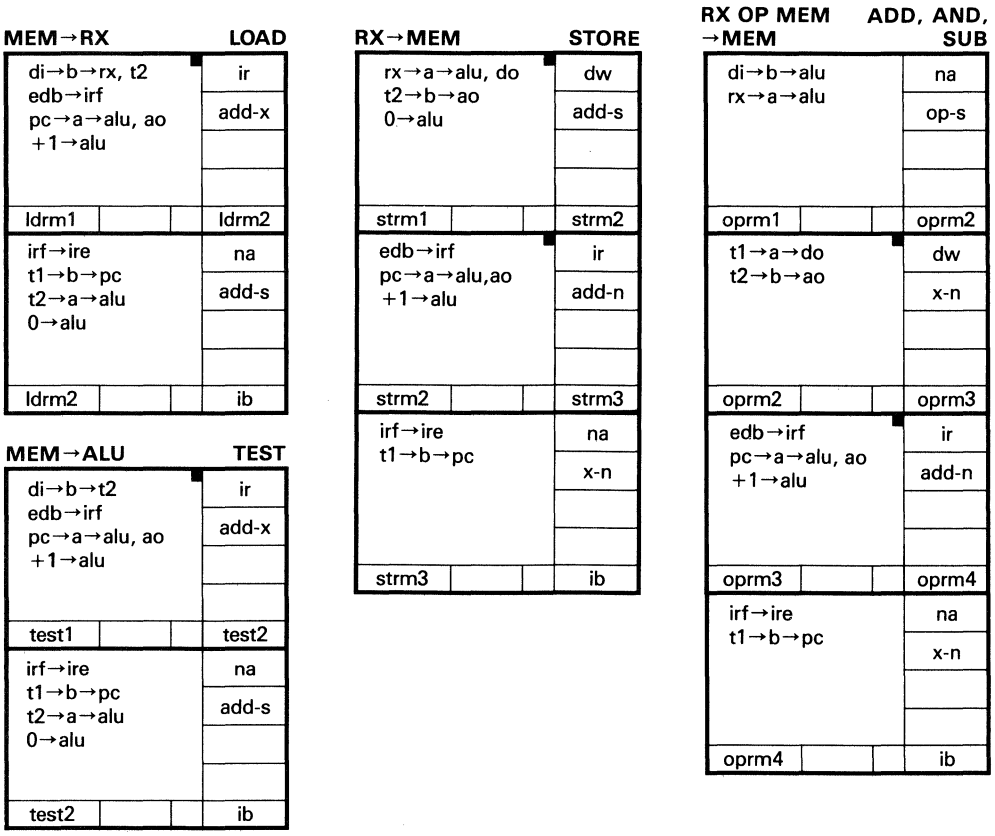


Figure 3.14 (continued)

the OR array. The PLA definition is derived from the labels in the flowcharts.

The label abbreviations are:

@	associated preceding quantity is an address
d	displacement
ADD	(for example) instruction using the sequence
MEM	MEMory
OP	OPeration
RX	source operand register
RY	address or operand register (see figure 3.3)

Execution Sequences for Register-to-Register and Special Instructions

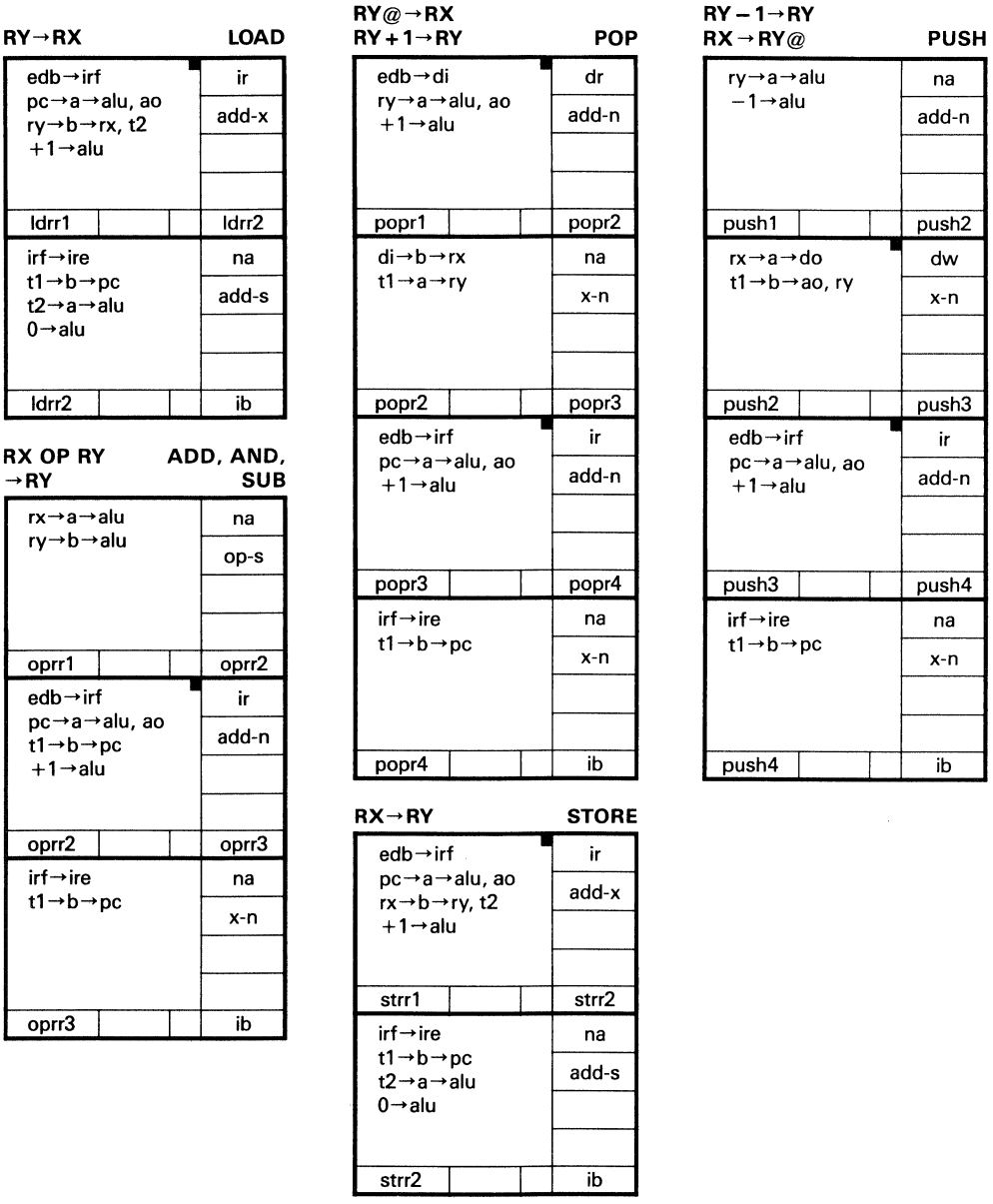


Figure 3.14 (continued)

2. **Access type.** This description says whether the controller is using the external bus for an instruction fetch or for a data read or write.
3. **ALU function and condition code setting.** The ALU function determines the operation for the ALU for a particular state. ADD, SUB, OR, and AND mean just what they say. OP means that the value in IRE determines what the ALU operation will be. The condition code setting tells whether a condition code is to be set.
4. **Duplicates.** This box is not used for the MIN CPU example. It is used by a flowchart drawing program to indicate how many other states contain exactly the same set of tasks. (We will use it later to help reduce control store size.)
5. **Page and location.** This box is not used for the MIN CPU example. It is used by a flowchart drawing program to place the associated flowchart box on a printer page. (For example, the Micro/370 flowcharts are twenty-five pages containing about a thousand states. Each state has its own page number and location coordinates assigned by the designer.)
6. **Next state transition.** The next state transition tells how the controller determines the next state. In a microcoded controller, the next state might be reached by a conditional branch, a sequence branch (a new address from an instruction decoder), or a direct branch (address from the current control word). For the MIN example:
   - BC (branch conditionally) denotes that the next control store address depends on the value of a condition code (generated by the ALU). A base address is supplied by the microword and altered (or augmented) by the branch condition.
   - SB (sequence branch) denotes a transition from an address mode sequence to the corresponding execution sequence for the current instruction.
   - IB (instruction branch) denotes a transition to the first state of the next instruction sequence. (In a microcoded controller, IB would tell the controller to access the control word at the control store address specified by the instruction decoder.)
   - State ID denotes a direct branch in the control store. The address of the next control word is in the current control word.
7. **Access width.** This box is not used for the MIN CPU example, but I would use it to indicate the size of the external bus

transaction. Micro/370 uses *w, h,* and *b* in this box to indicate external word, halfword, and byte accesses, respectively.

8. **Synonym.** This box is not used for the MIN CPU example. If several states have exactly the same tasks, then one is considered the original and the rest have the state ID of the original in the synonym box. (The duplicates box in the original state tells how many times the original state ID appeared in a synonym box.)

9. **State identification (state ID).** Each state has its own identifier. I use descriptive identifiers. For example, STRM1 is the state ID for the first state in the store-register-to-memory execution sequence. In a microcoded controller, the state ID will be a mnemonic representation of the control store address. Once you assign the control store addresses to the control words, you can use a program to translate the flowcharts' state IDs into control store address bit patterns. For the Micro/370 project, I used one program to assign the control store addresses and another to translate the flowcharts into the control word bit patterns.

Figure 3.14 shows some sample level 2 flowcharts with the above information. I used one method to reduce the number of states: sharing address mode sequences. A second method is to eliminate duplicate states at the ends of sequences by specifying a direct branch to a common sequence. This merges the ends of flowchart sequences. Do this by comparing the ending states of each sequence in figure 3.14 with all ending states below and to the right of the current sequence. Alphabetic organization of tasks, corner shadings, and access indicators make it easier to compare states. The result is figure 3.15 (pages 48–49), which has one-third fewer states than the flowcharts in figure 3.14. This is the most direct method of reducing controller size using flowcharts. When I did this for a CPU with hundreds of states (the MC68000), I wrote each state on a separate IBM card and alphabetized the deck. I then compared each card with the cards below it to find duplicate or similar states. Similar states have some tasks that differ. If you can make them the same without adversely affecting the associated sequences, you may eliminate some states.

When you merge the level 1 flowcharts to make level 2 flowcharts, consider moving operands into temporary locations early so later states in the sequence are more independent of the instruction parameters. (I did this in the ADD instruction example

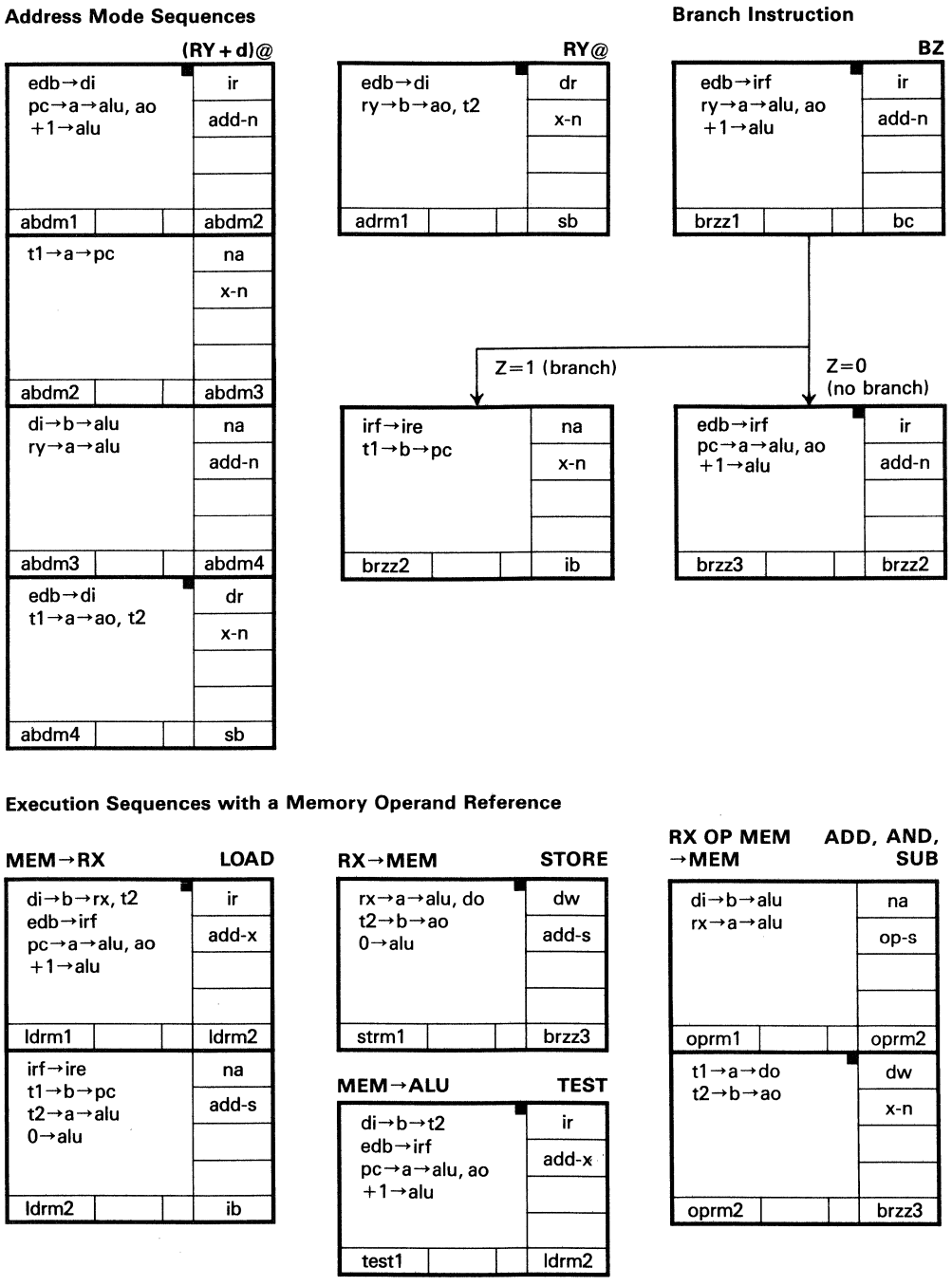


Figure 3.15 Merged level 2 flowchart examples (continues)

Execution Sequences for Register-to-Register and Special Instructions

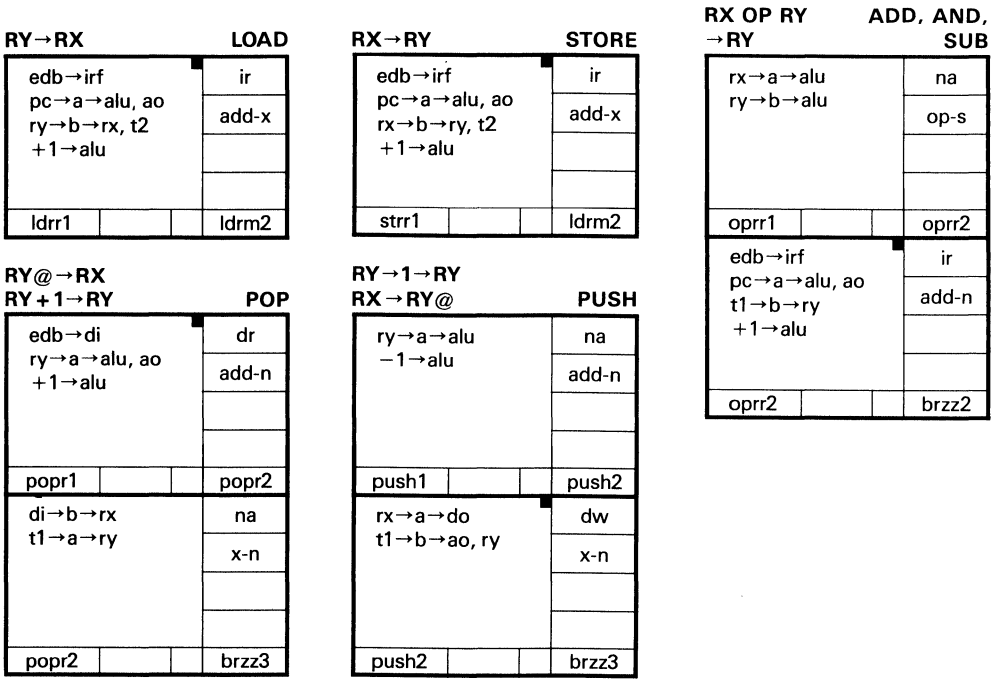


Figure 3.15 (continued)

for MIN.) Similar states occurring at other than the ends of the sequences cannot be merged. States adrm1, brzz1, and popr1 in figure 3.15 could be replaced by a universal state, but this state would have to exist for each sequence. It is possible to share common states at other than the ends of the sequences if some sort of microcode subroutine mechanism is provided.

Although a real CPU is much more complicated than the MIN CPU, the flowcharts would look just like those in figure 3.15. I implement my design from flowcharts such as these.

C H A P T E R

4

Implementing from Flowcharts

In this chapter I will explain how to implement a microprocessor from the flowcharts using the MIN processor example. Although I do not explain how to make the choice between (for example) a microcoded and a combinational design (see chapter 9 for the discussion of implementation methods), I will tell you how to implement the flowcharts for a microcoded, a PLA, and a combinational design.

Figure 4.1 is a block diagram of a simple microcoded controller, consistent with the function implied by the flowcharts. I show only enough detail for you to see the relationship between the controller and the flowcharts.

This is how the controller operates. An instruction is fetched (we'll worry about how some other time) and eventually placed in IRE. Translation of part of IRE's contents provides the control store address of the first word in the control word sequence for the instruction. Figure 4.2 shows the control store word format. Each flowchart state (see figure 3.15, pages 48–49) corresponds to a control word. The control word can specify register transfers (data and control registers), the ALU function and condition code setting, the source of the next control store address, and the next control store address.

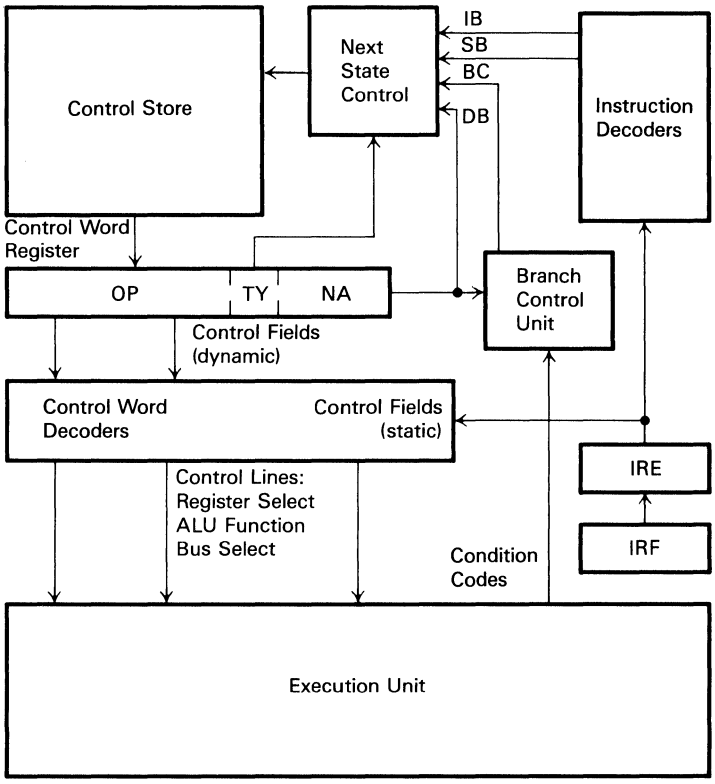
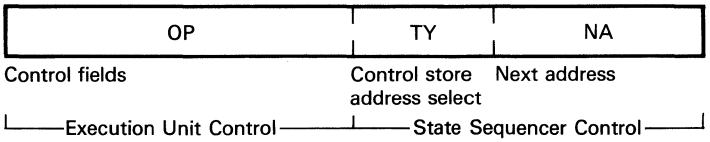


Figure 4.1 Microprocessor block diagram (microcoded controller)

The control word contains the address of the next control word for direct branches. For conditional branches, the next address would be modified by information from the execution unit (through the control store address modifier). For branches from address mode sequences to execution sequences or between whole instructions, the next address is a decoded IRE value (possibly modified by the control store next address [NA] field from the control word).

Each control word has fields that are decoded to drive the control lines in the execution unit and controller. The control word decoders (see figure 4.1) drive the control lines by mixing static information and timing signals with the control word fields. Static information does not change during the instruction execution and can go directly to the control word decoders. The register fields in a register-to-register ADD instruction, for example, do not change during instruction execution. The control



OP: Control Fields
Small fields of bits are decoded (by the control word logic decoders) to drive control lines in the execution unit and controller.

TY: Control Store Address Select
Next address (type) select

BC: Branch Conditionally—next control store address is NA modified by a condition code from the execution unit

DB: Direct Branch—next control store address is NA

IB: Instruction Branch—next control store address is from the control word decoders using IRE (for the next instruction)

SB: Sequence Branch—next control store address is from the control word decoders using IRE (for the next sequence to help execute the current instruction)

NA: Next Address
Next state (direct) address

Figure 4.2 Control store control word format

word tells when to move values to and from the registers, but IRE fields tell which registers will be used. That means ADD register 3 to register 5 shares the same execution sequence with ADD register 1 to register 7, for example. The fields in IRE that select the registers are the static information. (Remember that static information does not change during execution of an instruction. The register designators are static information. How they are used in each cycle is dynamic information. You might send the register contents to the ALU in the first state and store a result in the same register in state 3. When to do what is dynamic information; it changes with each state.)

In a simple microcoded controller implementation, each state in the level 2 flowcharts corresponds to one control word. (In a more complex controller, there may be more than one control word for each state. In the Motorola MC68000, for example, there is one control word for the execution unit and another for the state sequencer.) Since each state in the level 2 flowcharts maps to one word in the control store, the fewer states you have, the smaller the control store will be.

To personalize the control store, you must transform the flowcharts into control store bit patterns. Here are the transformations you need:

- The tasks become bits in the control fields (OP).
- The next state becomes the control store address select (TY) and next address (NA).

- The state ID becomes the location of the control word in the control store.

These transformations can be done on a computer. You translate all the states into control word bit patterns. Each of the control word bit patterns (representing a flowchart state) occupies a unique location in the control store. It is possible to have the computer assign control words to control store locations. For the Motorola MC68000 project, I assigned control store addresses, but for Micro/370, the computer assigned them. You may want to assign control store locations manually to make the control store address decoder smaller. Some control store addresses must be reserved for reset, interrupt, and other special sequences.

Figure 4.3 is a block diagram of a simple PLA controller. Note the strong similarity between the PLA controller in figure 4.3 and the microcoded controller in figure 4.1. I consider PLA controllers to be a variation of the microcoded controller. If the control

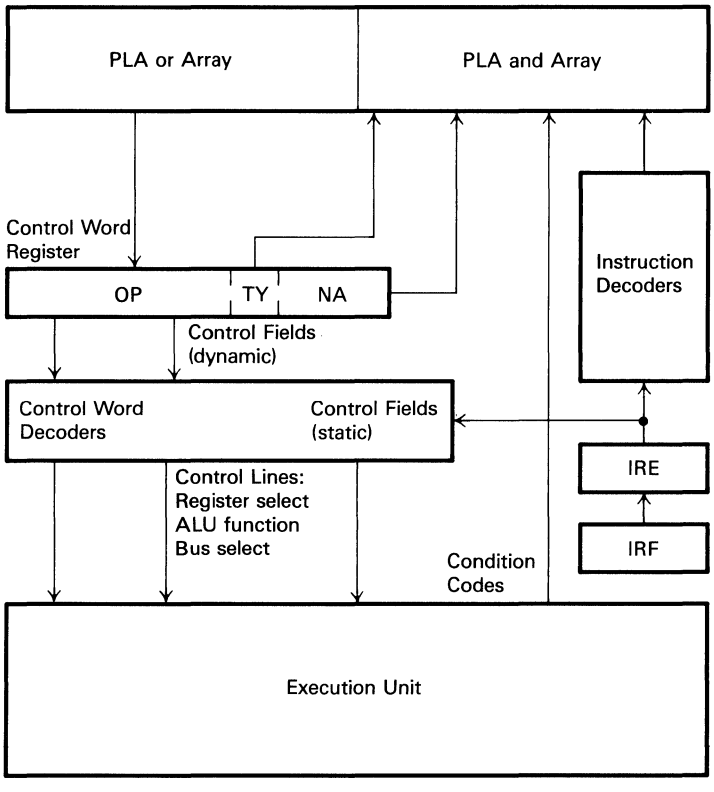


Figure 4.3 Microprocessor block diagram (PLA controller)

store address logic and the control store are implemented as an AND-OR PLA, the address logic would be the AND array and the control store would be the OR array.

Another way to see the similarity between a microcoded implementation and a PLA implementation is to consider the control store to be an orderly decode of an input address into a control word. If the control store address logic (address decoder, branch control unit, and multiplexer) of figure 4.1 produced the control word directly (instead of the address of the word in the control store), it would behave exactly like a PLA.

The flowcharts are used the same way except you may now be able to combine like states at other than the ends of sequences (provided the states can be made to lie logically next to each other for the AND decoder). Program the PLA OR array using the same flowchart transformations used for the microcoded controller. The PLA OR array contains the same bit patterns as the control store for the microcoded controller. Unused control store locations will be left out of the PLA. An apparent reduction in controller size may be possible using methods for splitting or folding a PLA. Although I will not discuss these methods here (they do not directly relate to using flowcharts), I do cover PLA folding briefly in chapter 8. I do not think it is a good idea to split or fold PLAs.

There are many ways to design a combinational controller (also called a combinatorial, random logic, or hardwired controller), but I will describe only one. First, design a state sequencer to duplicate the state transitions in the flowcharts. The flowcharts contain a complete state diagram. Techniques for converting state diagrams to state sequencers are known (see references at the end of this chapter), so I will not discuss them here. Next, make as many copies of the flowcharts as there are different tasks in the flowchart sequences (each line in a state box is one task). Each copy will be assigned to a different task. On the copy, mark all occurrences of the assigned task, then write an equation for the task using state IDs.

As an example, take the transfer of EDB to DI in figure 3.15. Assign this task to one copy of the flowcharts by highlighting all occurrences of the EDB-to-DI transfer. Write the equation for the EDB-to-DI transfer. If you chose to implement the controller from the level 2 flowcharts of figure 3.15, the equation for the EDB-to-DI transfer would be:

EDIN = abdm1 + abdm4 + adrm1 + popr1

EDIN is the name given to the control point (gate) controlling the transfer of EDB to DI. The real equation for the transfer would be much larger if all the flowchart sequences were available, but the technique is the same.

If you write the equations for unique states instead of individual tasks, you end up with the OR array from the PLA controller. (Lines duplicated in the OR array will not be duplicated here, but the state controller might be bigger.) This is because PLA design fixes the decode method for all terms (a two-level NAND-NOR or a three-level AND-OR-AND, for example), but combinational design allows the implementation of terms to vary individually. I could write equations for groups of lines, or I could group sub-expressions of the equations to reduce the logic. The flexibility of the combinational design probably is the source of some of the trouble it causes.

Relationship between Flowcharts and Hardware

I view flowcharts as a compact, precise description of hardware requirements. Implementing a controller from a flowchart description is a logic design procedure. Here are the steps I recommend for implementing a microcoded controller from flowcharts:

1. **Execution unit.** The execution unit is developed concurrently with the flowcharts. You add things as you need them to develop the mature execution unit.
2. **Instruction decoders.** The instruction decoders translate an instruction bit pattern to the control store address for the execution sequence. For the MIN processor example, I need two of these decoders. The first decoder translates the instruction bit pattern into the control store address for the appropriate address mode sequence. The first decoder provides the address labeled IB in figure 4.1. If there is no address mode sequence, the IB instruction decoder points to the execution sequence. The second decoder translates the instruction bit pattern into the control store address for the execution sequence. The second decoder provides the address labeled SB in figure 4.1. Instructions that do not have an address mode sequence do not use the SB instruction decoder. Only the last state of each address mode sequence selects the address from the SB instruction as the next control store address. If you don't go to an address mode

sequence, you don't use the SB address, so you don't care where it points.

3. **Control word format.** The control word format is derived from the flowcharts. In most design procedures, the control word format is determined before the microcode is begun. I think that's wrong. How do you know what the control words need to do? If you do not have flowcharts, you guess. That's how you end up programming the function you need using the control functions you are given, rather than always having the control functions you need. If you have flowcharts, you can let them tell you the required capabilities of the control word precisely.
4. **Control word decoders.** Having the control word, you can then design the control word decoders. These decoders combine the control word (dynamic) control fields, the IRE (static) control fields, and timing signals, to provide the gate control signals for all transfers in the execution unit and the controller. For example, one field in the control word tells when a register value goes from the register to a bus; an IRE field tells which register is to be connected to the bus; and timing signals say when the transfer takes place. These signals are combined in the control word decoders to produce a single gate control signal (or signal pair).
5. **Controller block diagram.** A lot of people think that the block diagram should come first, but I disagree. When you use flowcharts, you are making assumptions about how to accomplish what you need to do; you are adding detail. The user's manual for an architecture tells at one level how an instruction is executed. It tells you that the Add Register instruction adds the contents of two registers and puts the result in one of the registers. The flowcharts tell you at the next level of detail how that instruction is executed. They tell how the register operands are connected to buses and the ALU, when they go to the ALU, and when the result returns.

When I designed the flowcharts, I made assumptions about register control, buses, and all the things needed to execute the instructions. Now, as I do logic design, I collect all the assumptions and implement the design. I get just what I need to do the job—no more and no less. If you pick a controller (as most textbook authors would have you do), how do you know what it needs to do? If you do not have flowcharts, you guess. If you do have flowcharts, the assumptions for the controller are in the flowchart description. Part of the logic design procedure is to extract the assumptions and build the controller.

Sample Design Chronology

Appendix B is a design chronology for logic design for the Micro/370 project. I began doing flowcharts in January 1981. I did the initial design of the input and output multiplexers in December 1981. I did the initial instruction decoders and the control word format in January 1982. I did not derive the control word format until the flowcharts essentially were complete. I did the initial controller block diagram in August 1982—about twenty months after I began working on the design!

Appendix B is taken from my notebook of weekly activities for the project, so I am not just trying to remember how I did it or construct how I think it should be done. I am telling you exactly how it has been done. I kept the same kind of notebook of weekly activities when working on the MC68000 design. The event chronology is essentially the same for that project. (Events were more compressed at Motorola because the MC68000 is simpler than Micro/370 and because Motorola emphasizes microprocessor design. Here are the steps again:

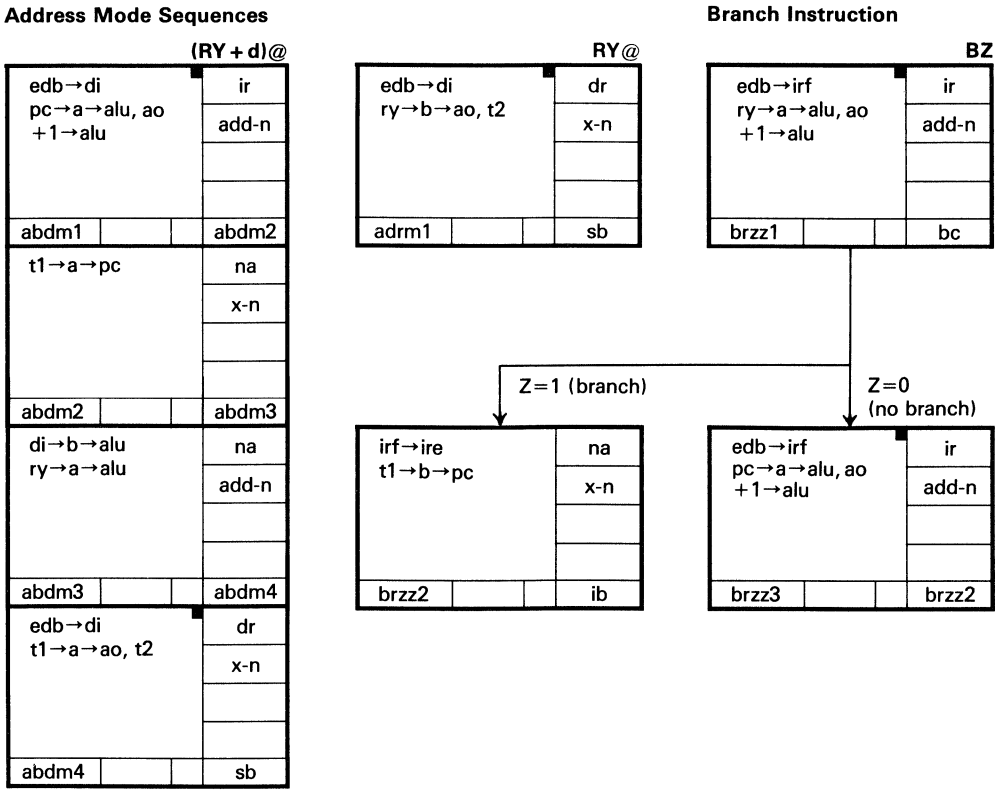
- execution unit
- instruction decoders
- control word format
- control word decoders
- controller block diagram

Sample Implementation Procedure

I repeat figure 3.15 here as figure 4.4 because I use it for the implementation examples. I show how to implement the instruction decoders, derive the control word format, and implement the control word decoders using the sample flowcharts for the MIN processor.

Execution Unit

You develop the execution unit concurrently with the flowcharts. You start with a proposal and modify it as required. When you finish the flowcharts, you also are done developing the execution unit. The last thing I defined for the Micro/370 execution unit was the input and output multiplexers. I looked at the flowcharts to see how the data and instructions got from the external data bus to the execution unit. In that microprocessor, there are byte, halfword, and word external bus transfers. Byte transfers can occur on any byte position on the external bus.



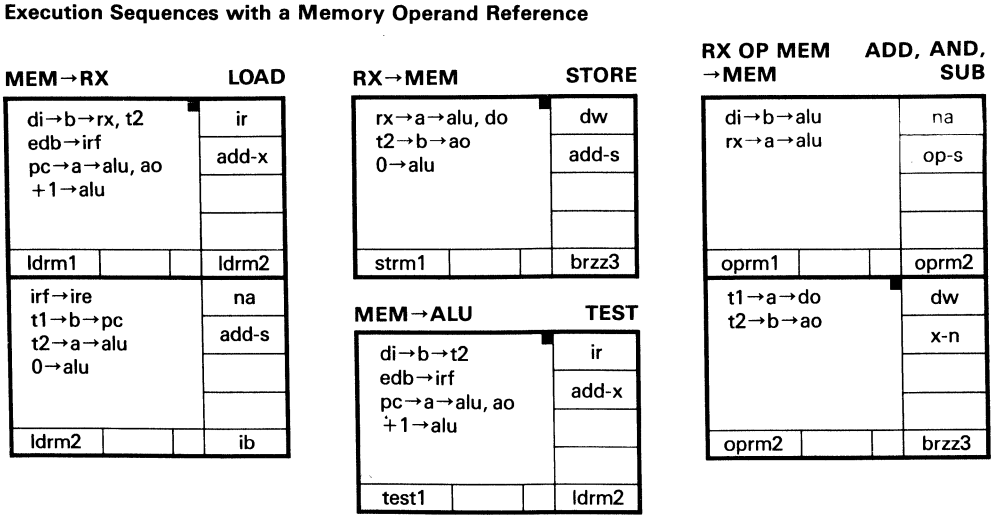


Figure 4.4 Merged level 2 flowchart examples (continues)

Execution Sequences for Register-to-Register and Special Instructions

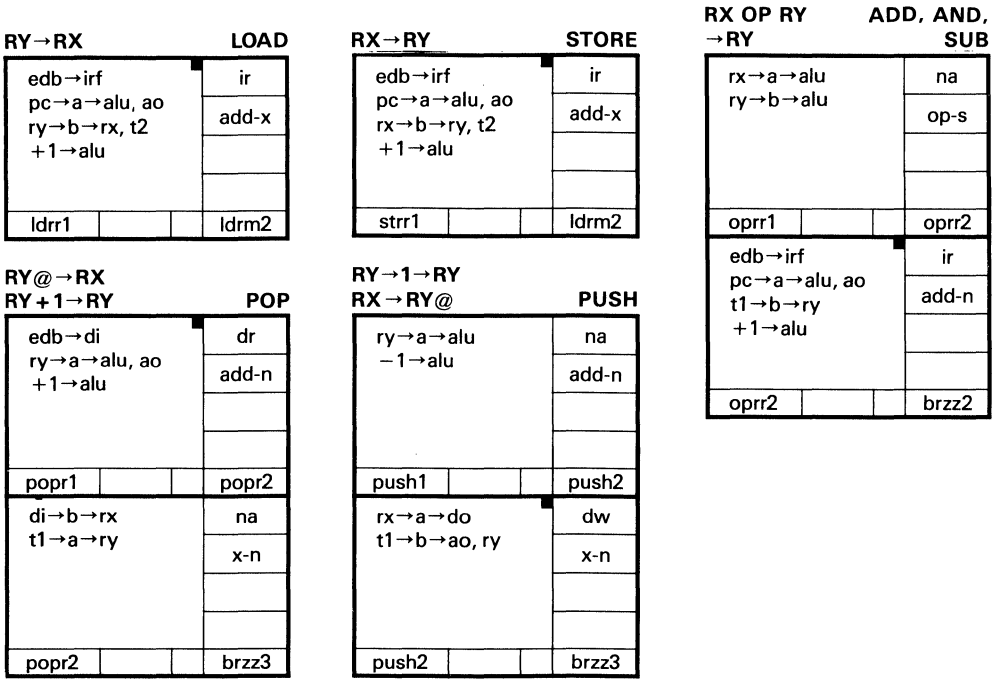


Figure 4.4 (continued)

The Micro/370 execution unit manipulates byte quantities only in the low-order byte position on the internal bus. I needed fancy multiplexers on the input and output connections to the external bus. The MIN processor does only word transfers, so it does not need the multiplexers.

Instruction Decoders

The MIN processor uses two instruction decoders. One decoder points to the first control word in an address mode sequence (if there is one), and the other points to the first control word in the execution unit sequence. The last state in any execution sequence shows IB in the next state block (see brzz2 and ldrm2 in figure 4.4). The IB instruction decoder points to the first control word in the next instruction sequence. The last state in any address mode sequence shows SB in the next state block (see abdm4 and adrm1 in figure 4.4). The SB instruction decoder points to the first control word in the execution sequence. For instructions with no address mode sequence, the IB instruction decoder points to the first control word in the execution se-

quence. (With no address mode sequence, the control word sequence and the execution sequence are the same. If there is no address mode sequence, it does not matter where SB points, since it is not used.)

Figure 4.5 shows sample instructions and the associated control word sequences, along with the IB and SB instruction decoder pointers. This shows how the flowcharts expect the instructions to execute. I designed instruction decoders IB and SB to map the path between instructions and from the address mode sequence to the execution sequence. One way to derive the mapping is to list all the instructions and then show the associated control word (if any) to which IB and SB should point. Another way to derive the mapping is to look at the first control word of each sequence and tabulate the instructions using that sequence (labels A and B) with the appropriate decoder. In practice, I use a third label on each control word sequence that tells

Instruction	Control Word Sequence	Next Control Word Address	IB Instruction Decoder	SB Instruction Decoder
POP	popr1	popr2	-	-
	popr2	brzz3	-	-
	brzz3	brzz2	-	-
	brzz2	-	abdm1	oprm1
ADD RX (RY + d)@	abdm1	abdm2	-	oprm1
	abdm2	abdm3	-	oprm1
	abdm3	abdm4	-	oprm1
	abdm4	-	-	oprm1
	oprm1	oprm2	-	-
	oprm2	brzz3	-	-
	brzz3	brzz2	-	-
	brzz2	-	oprr1	-
SUB RX RY	oprr1	oprr2	-	-
	oprr2	brzz2	-	-
	brzz2	-	adrm1	test1
TEST RY@	adrm1	-	adrm1	test1
	test1	ldrm2		-
	ldrm2	-	push1	-
PUSH				
.				
.				
.				

Figure 4.5 Example instruction execution sequences

me which instruction decoder points to the control word sequence. (Sometimes more than one decoder can point to the same sequence.) Figure 4.6 lists the control words to which the IB and SB decoders point for the flowcharts in figure 4.4.

The instruction mnemonics (such as AR, POP, L, and ST) correspond to bit patterns defined by the architects. The bit patterns should be listed in the user's manual. Even if you program the microprocessor in assembly language or FORTRAN using the mnemonics, the assembler or compiler translates the mnemonics into bit patterns. (You can store only ones and zeros in the memory, as these are the only values for which you can build logic.) The mnemonic control store addresses (such as oprr1, popr1, ldrm1, and strm1) also correspond to bit patterns (the real control store addresses). The instruction decoders translate the instruction bit patterns to the control store address bit patterns. If you build a PLA to implement the IB or SB instruction decoder, the bit patterns for the instruction mnemonics become the AND array, and the bit patterns for the corresponding control store address become the OR array (figure 4.6).

Control Word Format

I divide the control word into two sections based on function (see figure 4.2). The operation section, labeled OP, is composed of the fields for execution unit control. The next state section, containing TY and NA, contains the fields for state sequencer

IB Decoder Addresses	Instruction(s) or Address Mode	
abdm1	(RY + d)@	Address mode sequences
adrm1	RY@	
brzz1	BZ	Execution sequences (instructions **without** separate address mode sequences)
ldrr1	LR	
strr1	STR	
oprr1	AR, SR, NR	
popr1	POP	
push1	PUSH	
SB Decoder Addresses	**Instruction(s)**	
ldrm1	L	Execution sequences (instructions **with** separate address mode sequences)
strm1	ST	
oprm1	A, S, N	
test1	T	

Figure 4.6 IB and SB instruction decoders for MIN example

control. I divided the operation section into small fields that control the execution unit pieces. I did this by looking at the (logical) picture and simply assigning fields. To demonstrate the procedure, I have reproduced the MIN execution unit from figure 3.4 as a part of figure 4.7. Above that, I have allocated fields to the operation section of the MIN control word. There isn't much more to it than that. In chapter 8, I will show the much more complicated control words for Micro/370, but I used the same procedure to allocate the fields.

If two macros in the execution unit are never used at the same time, you might consider sharing the control field. This is done in a few cases for both Micro/370 and for the Motorola MC68000. Micro/370, for example, has one field that controls both the special Pack-Unpack unit and the shifter. The units do not have to have mutually exclusive use; you just have to be willing and able to do the extra control word decoding to separate the functions. Sharing control fields is not a good idea if the macros are not next to each other.

In a microprocessor, wire crossings are a big deal. You might save a field in the control word and waste more space crossing the signals to where you need them than you would have used by having an extra control field. In that case, you are trading area in the control store for area in the control word decoder. Carrying this to its extreme, let's say you save more and more fields in the control word. As you save fields, the control word decoder grows and the control word shrinks. When the control word is the same width as its input address, you don't need the control store. The result is a random logic implementation.

All I have done here is invent the format of the operation section of the control word. I have not assigned the number of bits to each field or made any bit pattern assignments. That's part of my logic design for the control word decoders: I look at the flowcharts to find out how many bits to assign to each field, and I use Karnaugh maps to help assign the best bit patterns to decode.

Control Word Decoders

Here is where I find out how many bits each control field needs and assign the bit patterns. I will show a few examples using the MIN processor flowcharts in figure 4.4. Here's the basic procedure:

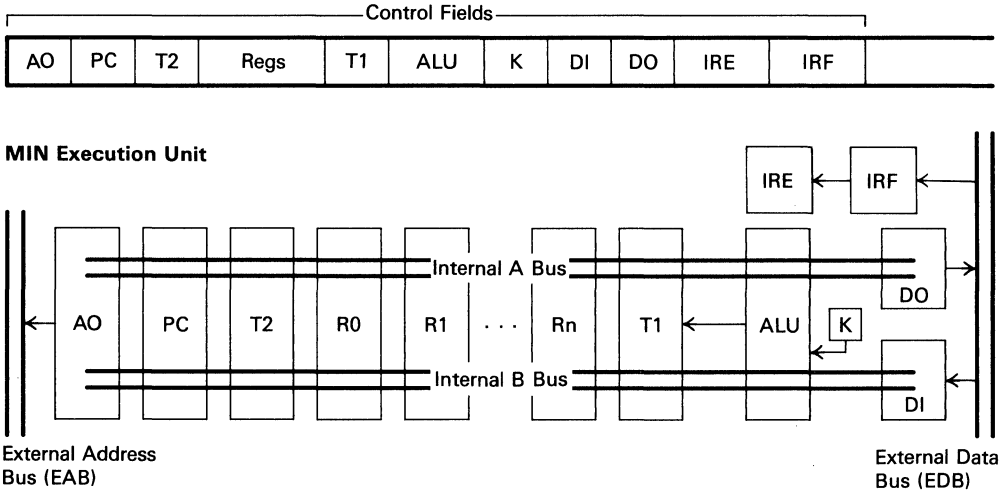


Figure 4.7 MIN control word (operation section) format

1. List uses of the macro.
2. Allocate bits.
3. Use a Karnaugh map to assign bit patterns.

That's it. Nothing difficult there. If you have your flowcharts in a computer data base, you can just call them up with the editor and collect all the occurrences of PC, T2, RX, or whatever. Once you have a list of all the different uses of a macro, you can assign the number of bits to the control field. If, for example, you did twelve different things with the Pack-Unpack unit, you would assign 4 bits to the control field. Don't forget to count "none" as one use. Some states do not use PC, for example, so you have to allow for those, too.

PC Control Example Look in figure 4.4 for all the ways PC is used. If PC goes to the bus, we don't care where it goes. If something writes to PC from the bus, we don't care where the data came from. We care only about controlling the paths into and out of the PC. To us, pc→a→alu looks the same as pc→a→rx. Here are the different uses of PC I have collected:

pc→a
a→pc (only one occurrence of this one)
b→pc
none

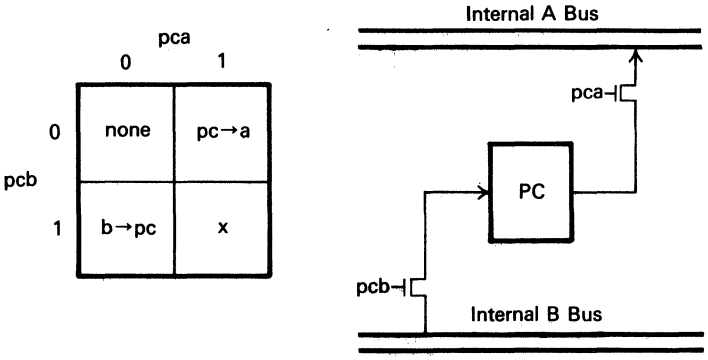


Figure 4.8 PC Karnaugh map and PC internal bus connection

Look again at the flowcharts in figure 4.4. There is only one occurrence of a→pc. I could just as easily write to PC using the B bus, so I can eliminate a→pc, thus simplifying the control logic. You probably will not be able to eliminate many of these uses in a processor with several hundred states, but I will do it here to simplify the example. With only three uses of the macro, I must assign 2 bits to the PC control field. Once I have the uses, I use a Karnaugh map to assign the bit patterns for the control field. Figure 4.8 is a diagram of the PC connection to the internal buses and the Karnaugh map for bit pattern assignment. One bit (pcb) will control an input connection to the PC from the internal B bus. The other bit (pca) will control an output connection to the internal A bus from the PC.

T2 Control Example I also looked in the flowchart examples of figure 4.4 for all occurrences of T2. Here's what I found:

t2→a
t2→b
a→t2 (only one occurrence of this one)
b→t2
none

This is about the same as the PC control example, as there is only one occurrence of a→t2. I can eliminate it, so I will. That leaves four uses (including "none"). I assign 2 bits to the control word field for t2. Figure 4.9 shows the internal bus connections for t2 and the Karnaugh map I used to assign the bit patterns. I can implement the logic for the control points using a 2-to-4 decoder. The 2 bits from the control word t2 field are the inputs. Three of the four output lines are connected (one each) to a control point; the fourth is "none" and is left unconnected.

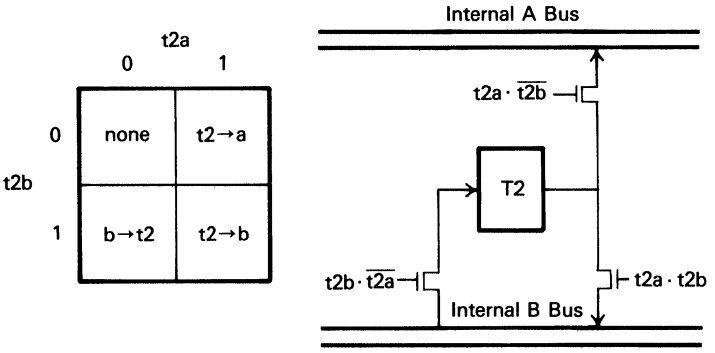


Figure 4.9 T2 Karnaugh map and T2 internal bus connection

Remember that the CPU logic works only with ones and zeros. When I want the CPU to do t2→b→alu, that's just what I put in the flowcharts. The CPU does not read the flowcharts, but I have to have somewhere to put the ones and zeros that eventually are transformed into the control signals on the gates.

The work I just did (in figure 4.9) tells me how to translate the flowchart statements into ones and zeros. The control store uses the ones and zeros of the T2 control field to drive the logic I just specified. That results in a signal on the appropriate control point (gate) in the execution unit. If I see t2→b→alu in the flowcharts, I should put 11 in the 2-bit T2 control field for that control word. If I do not use T2 in a particular state, then the T2 field for the corresponding control word should have 00 in it. What if a state contains two tasks using T2, such as t2→b→alu and rx→a→t2? Now what do I put in the T2 field? That case isn't listed among the ones I found in the flowcharts, so I didn't build hardware to do it. If you run into it, you have to change the state or change the hardware.

Register Control Example There's always a lot of confusion about register control, and it's easy to see why. The instruction has two ways to name registers. MIN calls one of the fields RX and the other RY (see figure 3.2). But the architecture contains only one set of registers: the programmer's register set. If there is only one set of registers in the architecture, there is only one set in the execution unit. RX and RY refer to the same set of registers.

In the MC68000 user's manual, for example, the writers refer to An, Dn, Rn, USP, SSP, SP, Ax, Ay, Dx, Dy, Rx, Ry, and Xi. I may

have missed one or two. Is it any wonder you get confused trying to design the control logic? There is only one set of seventeen 32-bit registers. Nine of the registers are called address registers, eight are called data registers, and two are stack pointers (USP, SSP, or SP). The stack pointers double as address register 7. Why the different names if there is only one set of registers? The variety of names is used in two ways. One is a logical grouping and the other is a physical grouping. The logical grouping separates the registers according to function. Mnemonics starting with A (An, Ax, and Ay) refer to any of the address registers, mnemonics starting with D (Dn, Dx, and Dy) refer to any of the data registers, and so on. The physical grouping separates registers according to which instruction field contains the register specification. The logical grouping tells you how to decode the instruction to tell whether you will be using an address or a data register. The physical grouping tells you where to find the field specifying the single referenced register in the logical group.

MIN has only one set of registers. RX means that you find the pointer to the referenced register in the first register field in the instruction. RY means that the register pointer is in the last field of the MIN instruction. It sounds as if all I need is a multiplexer to pick the register pointer from either RX or RY. Let's see. First, I collect all the flowchart references to RX or RY.

ry→a	rx→a; ry→b
b→rx	b→ry
ry→b; b→rx	b→rx; a→ry
rx→a	rx→a; b→ry
rx→b; b→ry	none

This collection of tasks looks (and is) more complicated than the set for T2. This time I had states with more than one task using the macro. Since all the tasks in the state are concurrent, I have to design the logic to be able to control two registers at a time. Next I look at the set of uses and draw how a typical register is connected to the internal buses. I did that in figure 4.10. I drew the connections to a typical register because I wanted to know the control points I would need.

R3 doesn't mean anything special; it's just one of the registers. Figure 4.10 tells me I need four lines from the control word decoders to control the registers. One line ($a \rightarrow r3/\overline{b \rightarrow r3}$) tells me whether the value written into the register comes from the internal A or the B bus. Another ($\rightarrow r3$) is active whenever a

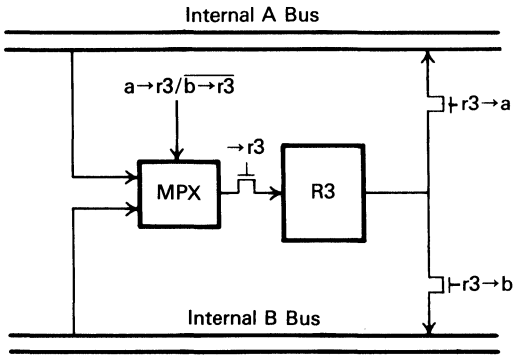


Figure 4.10 Connection of a typical register to internal buses

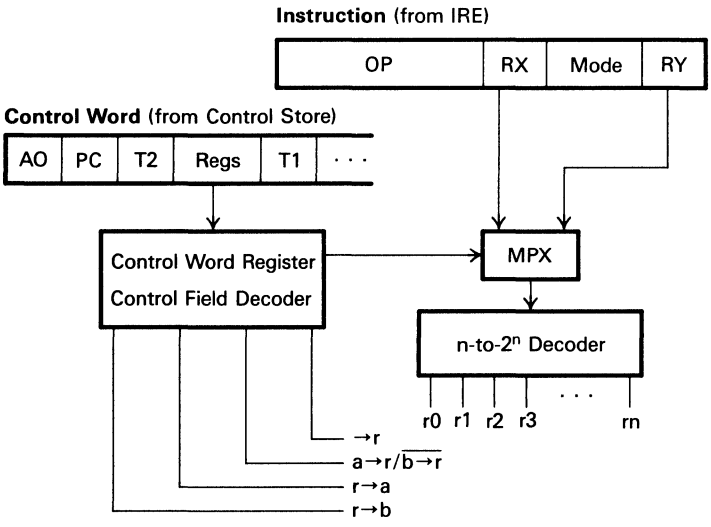


Figure 4.11 First proposed register control logic

value is written into the register. Two more (r3→a and r3→b) select the value in the register to one of the internal buses. The control word decoders should produce these four signals at the appropriate time. What about register selection? I intended to let the RX or RY field in the instruction select the register. The instruction is in IRE, so let's guess what the hardware might look like. Figure 4.11 was my first guess.

To get the control lines for a specific register, you AND the register selection line from the n-to-2^n decoder with each of the control lines from the control word register control field and use

that set of lines to control the register. (See figures 4.10 and 4.11.) It looks pretty good, but it won't work. From the collected set of register uses, I know that sometimes two tasks in a single state each use a register. I will not be able to derive a control signal for the multiplexer input selection in figure 4.11 because some states will want to select both fields. Let's try two decoders, associating one with the RX field and one with the RY field (figure 4.12).

To get the control lines for a specific register, you AND the register selection line from the n-to-2^n RX decoder with each of the control lines from the control word RX-control-field decoder. Do the same for the RY decoder and the control word RY-control-field decoder. Then OR these lines together. Here is what the control signals for register R3 look like:

$\rightarrow r3 = (b \rightarrow rx) \cdot x3 + (\rightarrow ry) \cdot y3$	load r3
$a \rightarrow r3 = (b \rightarrow rx) \cdot x3 + (a \rightarrow ry/\overline{b \rightarrow ry}) \cdot y3$	load from A
$r3 \rightarrow a = (rx \rightarrow a) \cdot x3 + (ry \rightarrow a) \cdot y3$	to A
$r3 \rightarrow b = (rx \rightarrow b) \cdot x3 + (ry \rightarrow b) \cdot y3$	to B

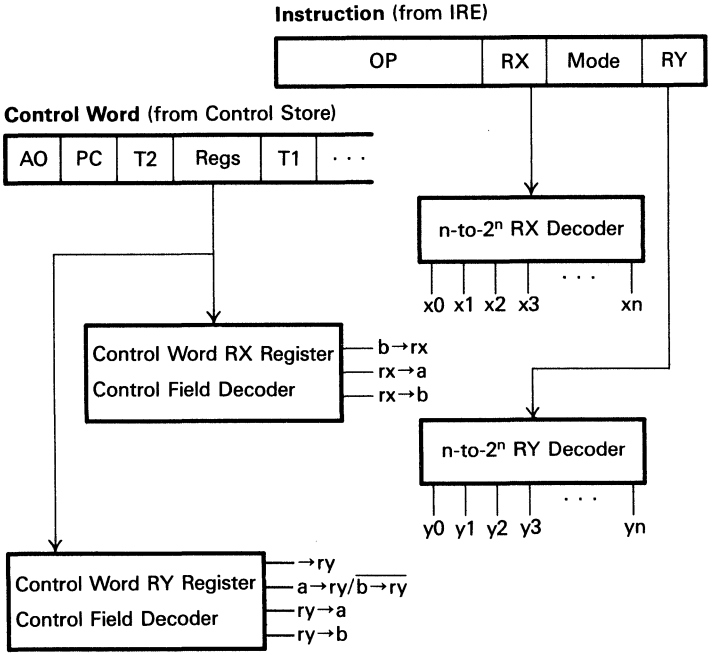


Figure 4.12 Second proposed register control logic

The occurrences in the flowcharts under Control Word States in figure 4.13 number ten entries. I need 4 bits for the control word field for register control. It's time to use the Karnaugh map to assign the bit patterns. From figure 4.12, I can see I need seven control lines from the control word decoders. Figure 4.13 helps relate the control lines I need to the tasks I found in the flowchart states. I also assigned the bit patterns to the control word register control field using a Karnaugh map. Each control word state must have a unique bit pattern in the control store (since each case implies a different combination of control lines). Figure 4.14 shows the bit patterns I assigned to the control store for the states I see in the flowcharts. The assignment of bit patterns to control fields is arbitrary. I try to assign patterns

Control Word States (What I see in the flowcharts)	**Control Lines** (Control signals from control word decoders)	**Control Lines** (Execution unit control points)
ry→a	ry→a	rx→a
b→rx	b→rx;→rx	ry→a
ry→b;b→rx	b→rx;→rx;ry→b	rx→b
rx→a	rx→a	ry→b
rx→b;b→ry	rx→b;→ry;b→ry	→rx
rx→a;ry→b	rx→a;ry→b	→ry
b→ry	b→ry;→ry	a→ry
b→rx;a→ry	b→rx;→rx;a→ry;→ry	b→rx
rx→a;b→ry	rx→a;→ry;b→ry	b→ry
none	none	

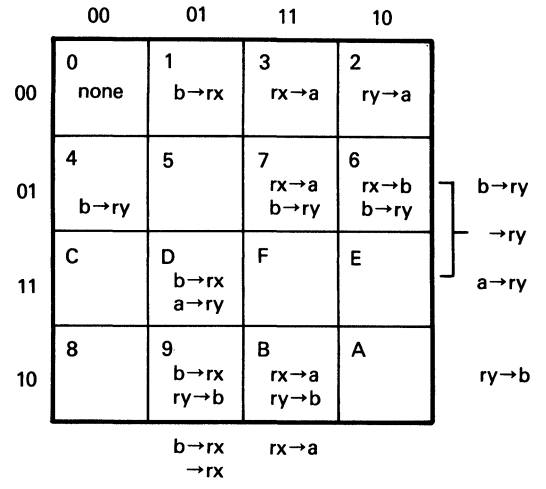


Figure 4.13 Relationship between the control word states and the control word register-field decoders

Control Word States (What I see in the flowcharts)	Control Field Bit Assignments (The bit pattern I put in the control word register control field)
none	0000
b→rx	0001
ry→a	0010
rx→a	0011
b→ry	0100
rx→b;b→ry	0110
rx→a;b→ry	0111
ry→b;b→rx	1001
rx→a;ry→b	1011
b→rx;a→ry	1101

Control Lines	Decoder Patterns (Control line is active any time the control field matches decoder pattern)
rx→a	xx11
ry→a	0010
rx→b	0110
ry→b	10xx
→rx	xx01
→ry	x1xx
a→ry	11xx
b→rx	xx01
b→ry	01xx

Figure 4.14 Control word register control field bit assignments

that reduce control word decoders. Figure 4.14 also shows the equations for the control lines. The equations show how the control field bit pattern is decoded to control the execution unit resource (in this case, the register file).

Let's try a few examples to see whether the control does what I want. State abdm3 (figure 4.4) contains ry→a→alu. From figure 4.14, I see the register control field for that state should contain 0010. If I put 0010 in the control word, what control lines are active? Compare the control word bit pattern (0010) with each of the decoder patterns in figure 4.14. The only pattern that matches activates ry→a. Exactly what I wanted.

See state push2 in figure 4.4. It contains both rx→a→do and t1→b→ao,ry. That case corresponds to a control word bit pattern of 0111 (figure 4.14) for the register control field. The following control lines are activated by the 0111 bit pattern:

Input Pattern	0111
Active Control Lines	
rx→a	xx11
→ry	x1xx
b→ry	01xx

Again, exactly what I wanted. The RX register is enabled to the A internal bus, whatever is on the B internal bus is selected to the RY input, and I get a load signal to RY.

How about rx→b→ry,t2 from strr1 in figure 4.4? That corresponds to a control word register control field of 0110. (For any

state, I should find only one bit pattern that fits.) Here are the active control lines for a control word pattern of 0110:

Input Pattern	0110
Active Control Lines	
rx→b	0110
→ry	x1xx
b→ry	01xx

You should have the idea now. For each flowchart state, a unique bit pattern is assigned to the control word register control field. The control word decoder translates the bit pattern into signals that activate the required control lines in the execution unit.

The register control I have shown works, but it is expensive. Control will be much cheaper if you:

- Assign one control word register-control-field decoder to each bus, rather than having a decoder associated with each register specification field, as in figure 4.12.
- Multiplex the input fields to the decoders for each bus.
- Do not allow transfers such as rx→*→ry (either bus).

Figure 4.15 diagrams the suggested register control logic.

ALU Control Example The flowcharts in figure 4.4 use the ALU in two ways. First, all instructions use the ALU for ADD to update the PC, and many instructions use the ALU for operand address calculation (also using ADD). Second, arithmetic and logical instructions use the ALU to perform the requisite operation (ADD, AND, OR, or XOR, for example). In the first case, the ALU operation is independent of the instruction context. In the second case, the operation I want from the ALU varies with the instruction op code. If the control word always specifies the ALU operation, then sharing the control word sequences among different instructions (such as ADD, AND, OR, SUB, and XOR) will not be possible because the control word must select different ALU operations for each. If the control word sequences are to be shared, the instruction register must participate in ALU function selection.

Figure 4.16 is the set of ALU controls from the flowcharts in figure 4.4. Figure 4.16 also contains a block diagram of the proposed ALU control logic and control points. The internal A bus is always an input to the ALU, and the output of the ALU goes to T1 for every ALU operation. The B side of the ALU gets

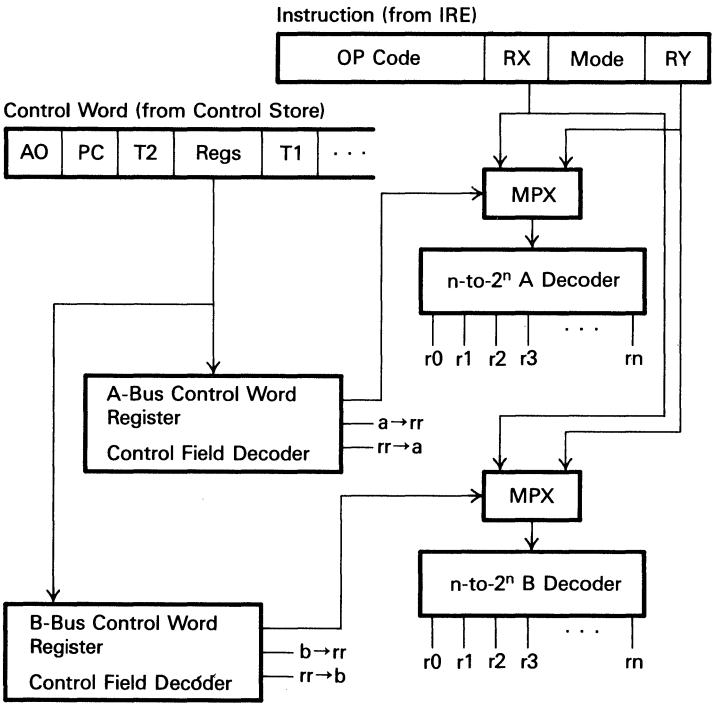


Figure 4.15 Suggested register control logic

either the value on the internal B bus or one of three constants used in the flowcharts. Either the ALU function is ADD or it is determined by an IRE field. The condition codes are either set or not affected. Six combinations of ALU controls occur in the flowcharts (including "none," which preserves the value in T1). I need three control word bits to control ALU operation. Figure 4.17 includes the Karnaugh map, the control word bit assignments, and the control line equations for the ALU control example.

State Sequencer

The MIN state sequencer is made up of the branch control unit and next state control.

Branch Control Unit

The branch control unit (figure 4.1) modifies the control store next address. State sequencing depends on conditions in the execution unit. This supports conditional branch instructions such as Branch on Zero (see BZ in figure 4.4). The microcode itself

Control Word States (What I see in the flowcharts)	Control Lines (Execution unit control points)
a→alu; +1 →alu;add-n;alu→t1	load t1
a→alu; b→alu;add-n;alu→t1	load ccr
a→alu; 0→alu;add-s; alu→t1	add/op select
a→alu; b→alu; op-s; alu→t1	alub input select (2 bits)
a→alu; −1 →alu;add-n;alu→t1	

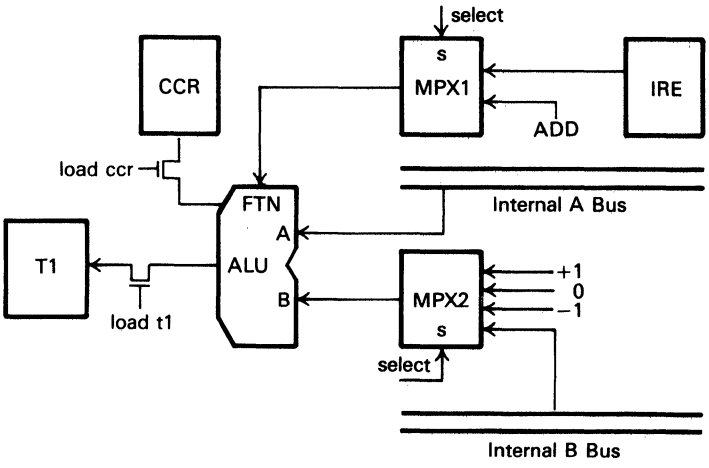


Figure 4.16 Proposed ALU control logic

Karnaugh Map for ALU Control Assignment

	00	01	11	10
0	0 none	1	3	2 b→alu add-n
1	4 0→alu add-s	5 +1→alu add-n	7 −1→alu add-n	6 b→alu op-s

Control Word States (What I see in the flowcharts)	Control Field Bit Assignments (What's in the control field)
a→alu; +1→alu;add-n;alu→t1	101
a→alu; b→alu;add-n;alu→t1	010
a→alu; 0→alu;add-s;alu→t1	100
a→alu; b→alu; op-s;alu→t1	110
a→alu; −1→alu;add-n;alu→t1	111
none	000

Control Lines (Execution unit control points)	Decoder Patterns (Control line is active any time the bit pattern in the control field matches decoder pattern)
load t1	xxx i000 (all except 000)
load ccr	1x0
add/op select	110
alub input select B	x10

Figure 4.17 Proposed ALU control word bit assignments

needs conditional next states for branches in the microcode. For example, if there is a multiply instruction and no hardware multiplier, the implementation of the multiply instruction uses a conditional branching algorithm. Instructions do not "see" these branches (except as different execution times for the instruction). Also, if the CPU checks for arithmetic overflow, you can achieve this with a conditional branch in the microcode.

Use the flowcharts to help design the branch control unit. It is just like other parts of the design: You assume whatever you need while you are working on the flowcharts, then you extract the specification from the flowcharts and build the unit. In the MIN example flowcharts, I have shown only one conditional branch. That is not enough to be a convincing example. I will assume that there are four types of conditional branches used in the instructions. They are Branch on Zero (BZ), Branch on Overflow (BV), Branch Negative (BN), and Branch on Carry (BC). This set of branch conditions (zero, overflow, negative, and carry) also is a sufficient set for the microcode (whose branches are not "seen" by instructions).

Here's the scheme. The control word (TY field, shown in figure 4.2) says "knock, knock" to the branch control unit. Some other part of the control word will choose the branch condition of interest. This part of the control word says to the branch control unit, "Is *this* branch condition (e.g., overflow) there?" The branch control unit just says yes or no. Since there are four possible branch conditions (using the Z, V, N, and C condition code bits from the ALU), I need two bits to choose the branch condition. I do not need a "none" in this case because the control word (TY field) will not select the branch control unit's output unless there is a conditional branch. Because each of these branch conditions is indicated by a single-bit value, I need only a 1-bit output from the branch control unit. A simple way to implement this branch unit is to use the ALU condition bits (Z, V, N, and C) as inputs to a 4-to-1 multiplexer and to let the control word select the branch condition. The branch control unit output bit (BC) is substituted for the low-order bit from the NA field.

Where in the control word is the field that selects the branch condition? I could add one (2-bit) field to the control word format. But that would add 2 bits to every control word, even though branch-type control words are not used very often. We don't want to do that. Since one control store address bit comes from the branch control unit, there is an extra bit in the NA field

whenever the conditional branch is specified (by the TY field). I just need one more bit. Where can I get it?

I can think of two simple solutions, both involving bit-robbing. The first is to fix one of the control store address bits for all control word branches. If I fix the high-order bit at zero and let the branch control unit provide the low-order bit, then there are 2 free bits in the NA field (a bit at each end) that can be used to select the branch condition. This means that all branch targets in the control store must have an address with a high-order zero. The second solution is to make the branch control unit have a 2-bit output. If the branch control unit provides 2 bits, there are 2 free bits in the NA field again. I don't even have to make the branch control unit any more complex. I could use its single output line for the 2 low-order address bits for the next control store address. Both methods restrict addresses for control words that are branch targets. That is, they complicate control word "placement" in the control store.

Next State Control

Next state control for MIN is a simple multiplexer. I have four sources for the next control store address: two from the instruction decoders, one from the branch control unit, and one from the control word. The control word (TY field) selects the next control store address (figure 4.18). The next state control for MIN is simpler than the next state control for a commercial microprocessor. Next state control for MIN implements four next state sources, while Micro/370, for example, implements nine. MIN does not implement interrupts or system reset signals.

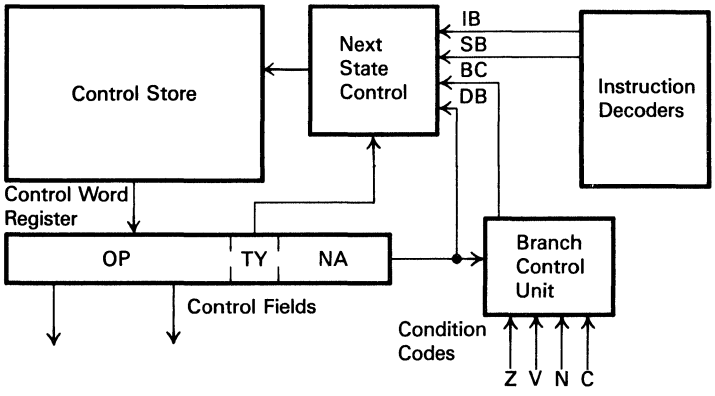


Figure 4.18 Control store branch control and next address select

Interrupts and system reset signals increase the complexity of next state control. For MIN next state control, the control store address sources are mutually exclusive—you always know which address source to use for the next control store address. Next state control implementing interrupts and resets must rank competing address sources to select the next control store address source. (Instruction prefetching, the potential for page faults, and architectural requirements will complicate next state control—as you will see.)

Summary

The format of all the control word fields is determined as above. Start at the left side of the execution unit and define the control word fields for each macro until you get to the right side. Use the flowcharts to tell you what you need. The execution unit tells you the number of fields, while the flowcharts determine how many bits to assign to each field. Draw a block diagram of the macro and the control for the macro to determine number of control lines. Then use a Karnaugh map to assign control field bit patterns in a way that minimizes the control word decoders.

The flowchart method transforms the English-language specification of the CPU into a formal description of how the CPU behaves. Use the flowcharts to construct only the hardware you need to implement the specified CPU. Derive the control word format from the flowcharts. You need not guess what the requisite capabilities of the controller must be or what the desired format and capabilities of the control word are.

References

Christopher R. Clare, *Designing Logic Systems Using State Machines* (New York: McGraw-Hill, 1973).

William I. Fletcher, *An Engineering Approach to Digital Design* (Englewood Cliffs, New Jersey: Prentice-Hall, 1980).

Zvi Kohavi, *Switching and Finite Automata Theory* (New York: McGraw-Hill, 1970).

John B. Peatman, *The Design of Digital Systems* (New York: McGraw-Hill, 1972).

Charles H. Roth, Jr., *Fundamentals of Logic Design*, 2d ed. (St. Paul, Minnesota: West Publishing Co., 1979).

David Winkel and Franklin Prosser, *The Art of Digital Design* (Englewood Cliffs, New Jersey: Prentice-Hall, 1980).

CHAPTER

5

How a Microprocessor Works

In this chapter I will discuss, in some detail, how a microprocessor works. This procedure will be like taking an auto engine apart and examining it piece by piece. You will get to see a real crankshaft—and believe me, it doesn't matter that it's from a 289 Ford. Seeing the real thing gives you a reference point for asking questions about why things are done a certain way. I will talk about problems in designing logic for the IBM System/370 architecture. It would be too bad if your reaction is "that's for the stupid 370 . . ." because I think the System/370 is, in fact, an outstanding architecture. Study the solutions. I will slip in explanations of IBM Micro/370 and Motorola MC68000 parts to prepare you for chapters 6 and 7.

Figure 5.1 is a diagram of a microprocessor. Assume that the microprocessor is running a program. I view instruction execution as a composition of three parts: fetch, decode, and execute. The simplest CPU would work serially: fetch the instruction, decode it, execute it, then fetch the next instruction. In more complicated CPUs, fetch, decode, and execute happen at the same time. While the CPU is executing the current instruction, it is decoding the next and fetching the one after that. Let's see what is happening

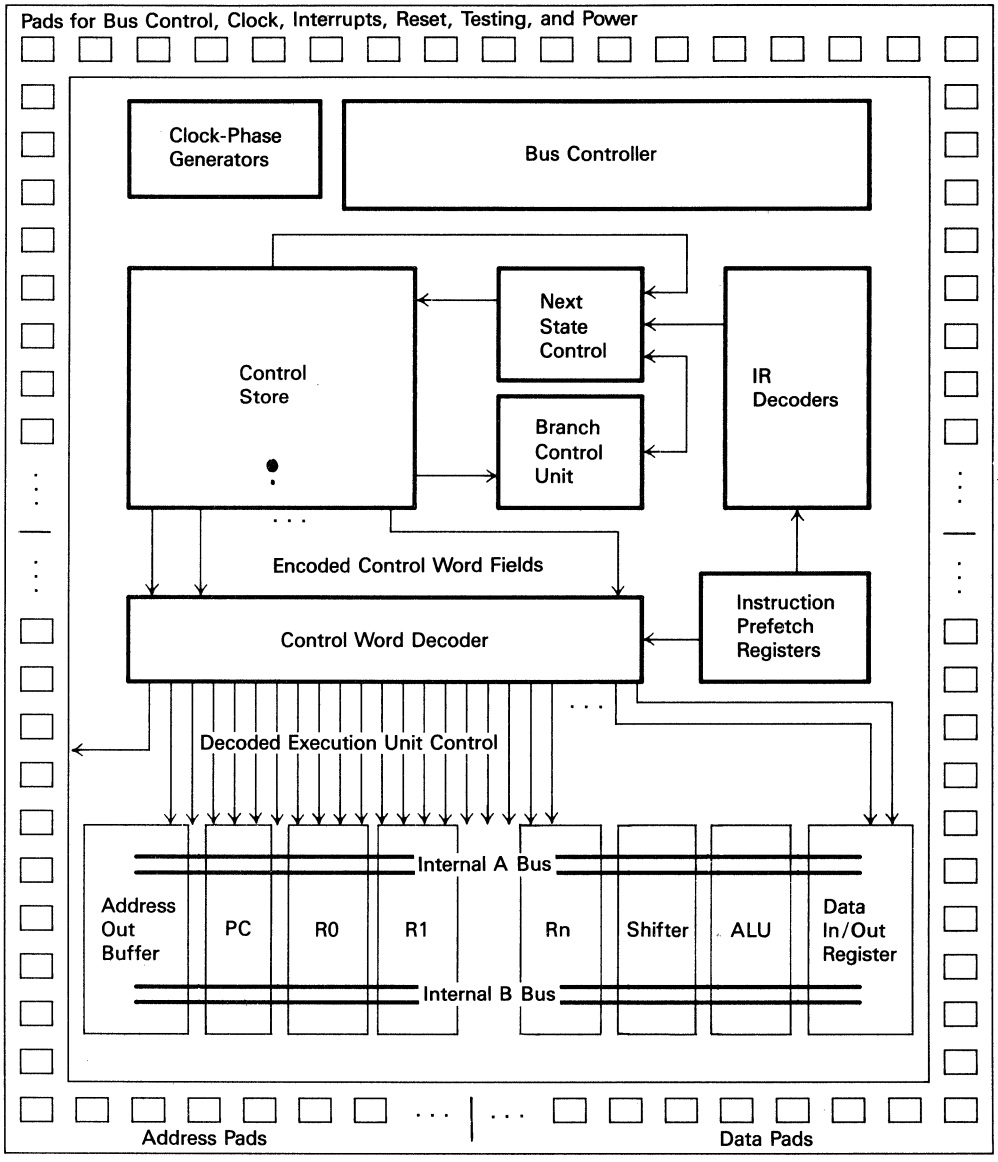


Figure 5.1 Microprocessor chip

while a program is running. Assume that the microprocessor has just finished decoding the current instruction.

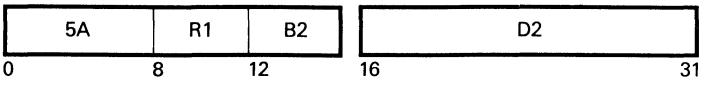
The instruction format is shown in figure 5.2 (which appeared earlier as figure 2.5). 5A (hex) is the op code for ADD. The R1 field designates a general register where the first operand is found and where the result will be stored. The B2 field designates the general register to be used as a base address register. The D2 field designates a displacement value (halfword) to be added to the contents of the base register to form the address of the second operand.

Here's what has to happen to get the microprocessor to the same point in the next instruction.

1. Fetch the remaining instruction halfword.
   The incremented PC value is placed on the address pads, and the controller waits for the halfword. The returning halfword is saved in a temporary register.
2. Calculate the operand address.
   The register designated in the B2 field is added to the contents of the D2 field.
3. Fetch the operand.
   The calculated operand address is sent to the address pads, and the state sequencer waits for the operand halfword to return. It is saved in a temporary register.

ADD

A R1,D2 (B2)



The second operand is added to the first operand, and the sum is placed in the first-operand location. The operands and the sum are treated as 16-bit signed binary integers. The first operand is in the register specified by the R1 field. The second operand is in memory. The address of the second operand is formed by adding the displacement specified by the D2 field to the contents of the base register specified by the B2 field.
An overflow causes a program interruption when the fixed-point overflow mask bit is 1.

Resulting Condition Code	Program Exceptions
0 Sum is zero	Access (fetch)
1 Sum is less than zero	Fixed-point overflow
2 Sum is greater than zero	
3 Overflow	

Figure 5.2 The ADD instruction

4. Add.
   Add the contents of the register designated by the R1 field to the contents of the temporary register. The answer is held in the ALU.
5. Store the answer.
   The result is sent to the register designated by the R1 field.
6. Update the PC.
   The PC is incremented to point to the next instruction halfword. It is saved during the next state.
7. Fetch the first halfword of the next instruction.
   The new PC value is sent to the address pads, and the state sequencer waits for the instruction halfword.
8. Find the control store address of the next instruction's control word.
   The next instruction halfword is sent to a register at the input to the instruction decoders, which are called IR (instruction register) decoders.
9. Branch to the next instruction control word sequence.
   The IR decoders produce the address of the next instruction's control word sequence. The last state in the control word sequence for the ADD instruction changes the next state control (figure 5.1) to choose the next control store address from the output of the IR decoders.

Each of the states in the sequence is represented by a control word in the control store. The control word format is shown in figure 5.3. Each control word contains encoded control patterns for control of the execution unit, bits to select the next control store address, and a field for the next control store address. In the control word sequence for the ADD instruction, each of the control words except the last contains the address of the next control word. The last control word selects the address at the output of the IR decoders.

In a commercial microprocessor, many of these steps overlap—that is, occur at the same time. It takes one cycle to decode the instruction (make the control store address); only the IR decoders are used. During that time, you can use the execution unit and the external bus. I use the external bus to read or write data

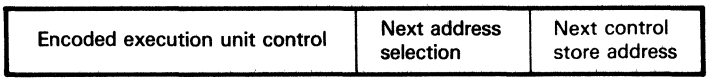


Figure 5.3 Control word format

State	Execution Unit	Decoder	External Bus
1			Read instruction halfword
2	ALU = D2 + (B2)		
3			Read operand halfword
4	ALU = (DI) + (R1)		
5	R1 = (ALU)		
6	ALU = (PC) + 2		
7	PC = (ALU)		
8			Read instruction halfword
9		IR	

Figure 5.4 ADD instruction execution with no overlap

State	Execution Unit	Decoder	External Bus
1	ALU = D2 + (B2)		Read instruction halfword
2			Read operand halfword
3	ALU = (PC) + 2		
4	PC = (ALU)		
5	ALU = (DI) + (R1)		
6	R1 = (ALU)	IR	Read instruction halfword

Figure 5.5 ADD instruction execution with overlap

or prefetch another instruction halfword. I use the execution unit to put a result in a register or to update the program counter. Overlapping operations speeds instruction execution (I use processor resources concurrently rather than serially).

Figure 5.4 shows the resources of the microprocessor and the states of the ADD instruction, without overlap. See how sparsely the resources are used. The idea of overlapping is to use the independent resources concurrently to complete the job in fewer states.

Figure 5.5 shows how the ADD instruction can be overlapped. I made subtle changes in the way the microprocessor executes instructions. The calculations are all the same, and I still have the same number of external bus accesses, but the assumptions are different. The instruction halfword being fetched in state 1 cannot be the second halfword of the ADD instruction, because I assumed that halfword was in the microprocessor when I used the displacement in D2. The halfword fetched in state 1 is the first halfword of the next instruction. The halfword fetched in state 6 is either the second halfword of the next instruction or the first halfword of the instruction following the next instruc-

tion. This works okay for a series of ADD instructions (as in figure 5.5) because I need to have the second halfword in the microprocessor at state 1 and the sequence for the ADD puts it there.

This design looks better, but it still is not complete. There are two instruction accesses in the sequence, but the instruction address (PC) is only incremented once. Further, there is no place to save the extra instruction halfword between instructions. These must be added. There isn't any way to accomplish the PC increment without increasing the number of states in the sequence—unless I add some hardware for that, too. I have already said that a commercial microprocessor could do the instruction in five states, so let's keep working on this till we get there.

First, I need a more accurate representation of the execution unit. I'm going to change notation, too. It's getting too difficult to keep track of everything that is going on in a single cycle. I will use a box to show one state. Inside the box, I will list the tasks that occur during that state. These tasks occur at the same time, so I list them in alphabetical order for convenience. In our microcoded controller, each state will be represented by one control word. A sequence of states will implement an instruction. Figure 5.6 is an execution unit with enough detail to see what the processor is doing to execute the instruction. Figure 5.7 is the ADD sequence of figure 5.5 using the new notation.

This control word sequence appears to do everything in five states. I added hardware to improve performance. Now there is

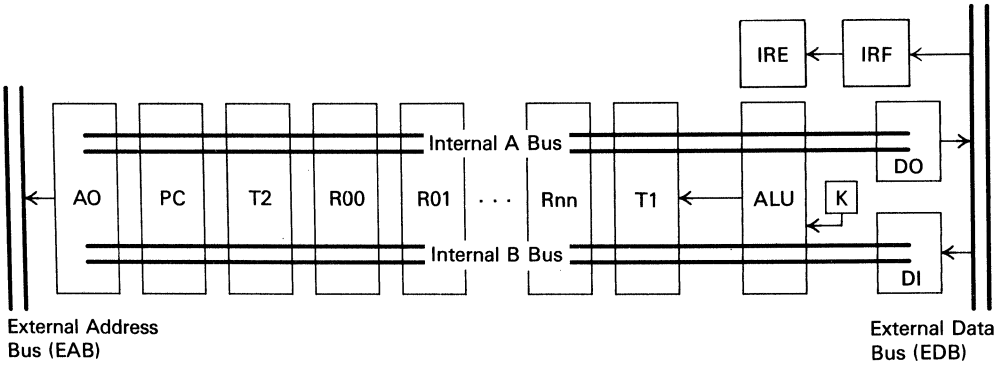


Figure 5.6 A more detailed execution unit

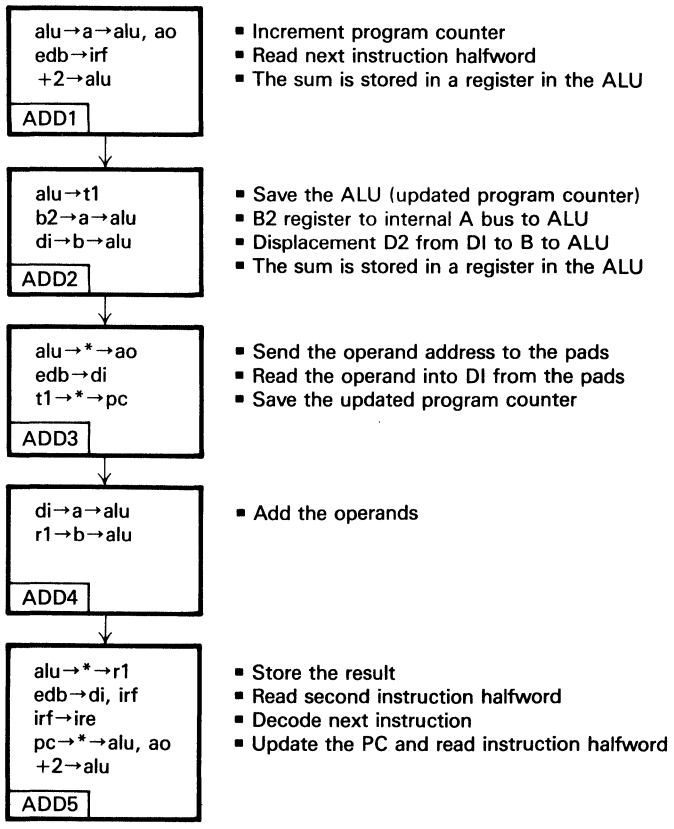


Figure 5.7 ADD instruction execution with overlap (state notation)

a new instruction register for fetch (IRF) to hold the next instruction while the current instruction is being executed. The ALU is directly connected to a temporary register to allow simultaneous calculation and saving of the previous result. Am I done? No. The problems are subtler.

Look at state ADD5, where I am fetching the halfword following the next instruction, decoding the next instruction in the IRE, incrementing the program counter, and saving the result of the ADD. If I change the value of IRE, the answer for the ADD instruction might get put in the wrong place because the IRE holds the pointer to R1. State ADD2 also looks funny. The ALU is used as a source (to T1) and as a destination. How do I know what value of ALU is stored in T1? To solve these problems, I introduce clock phases.

Internal Clocking

I said that all the tasks in a state happened at the same time. I lied. They can't. It depends on how closely you look. From an external view, everything in the state appears to happen at the same time. (If you are outside the chip, you can observe only what is happening at the pads.) Inside the chip, you have to have a clocking scheme to regulate events. One common clocking scheme uses a four-phase clock.

During phase 1, the source register is gated to the internal bus. During phase 2, the signal on the internal bus is amplified and broadcast the length of the bus. During phase 3, the signal on the internal bus is gated to the destination. In phase 4, the bus is returned to a neutral state. In state add3, for example, the register T1 is sent to one of the internal buses in phase 1. In phase 2, the contents of the register are amplified and broadcast down the bus. During phase 3, the gate to the PC register is opened and the signal on the bus overwrites the contents of the PC.

In state ADD4, the register DI and the register designated by the R1 field of instruction are gated to buses B and A, respectively, during phase 1. During phase 2, the contents of the registers are amplified and broadcast down the bus. During phase 3, the inputs to the ALU are opened, and the ALU begins to operate. During phase 4, the result of the ALU operation is saved in the ALU output register, and the resulting condition codes are sent to the condition code register.

That's the way it's supposed to work. Actually, however, the ALU is part of the critical path in the execution unit, so it is always connected to the internal buses and always operating, but the result is saved (at phase 4) only for the states that use the ALU. It gives the ALU a head start and explains why the register associated with the ALU is at the ALU's output and not at the ALU's input.

That's the clocking scheme—almost. There is one more piece—memory. I put addresses out, but I don't know when the data will come back. The clocking on the chip has to allow for delays from the external bus. I do this by adding another clock phase. I call it phase 4 prime. The only thing it does in the execution unit is clock the inputs to IRF and DI (if the transfer is enabled). This lets me compute the result once and wait for the memory data

for as many clocks as is necessary—without affecting other parts of the execution unit state.

Let's look at the internal timing of the figure 5.7 states to see whether everything is working okay. In state ADD1, the PC will reach AO at phase 3. AO is the buffer that drives the pads. The instruction returned by the memory on the EDB will arrive at phase 4 or (some) phase 4 prime. In ADD2, the value being saved in T1 is the updated PC. The ALU-to-T1 transfer must be a phase 1 transfer. The transfers in ADD5 are a problem. The next instruction is in IRF and must be moved to IRE before the next halfword is loaded at phase 4 (or phase 4 prime). But the register pointer (R1 field of the instruction), used in phase 3, depends on the contents of IRE. The problem in ADD5 is still there. I am changing IRE in phase 1, but I need the R1 field in phase 3. I will postpone fixing this problem until I discuss controller timing.

Timing between the Execution Unit and the External Bus

Now that it looks as if I have the timing down pretty well, here is a rule of thumb I use for today's technology. I can execute two internal states (microcycles) for every external bus access. (Two CPU cycles per bus cycle.) This means I can put the address on the pads in one state and look for the returning data at the end of the next state. I call these split cycles. The people who buy your chip can measure performance only at the pads. The more processing you hide inside the memory references, the faster the chip looks. Two microcycles per external bus access is a rule of thumb I have used since 1977; the rule is still valid.

Instruction Execution Time

To assess the execution time of the instruction, I count the state as one unit if it has no external reference and two units if it has an external reference. I call these units microcycles. The instruction sequence in figure 5.7 takes eight microcycles to complete (even though it uses only five control words). Split cycles count as one unit each because they distribute the memory delay over two internal states. Let's see how I can improve the performance of the processor by taking advantage of split cycles. Figure 5.8 is the same ADD instruction but uses split cycles to improve performance.

The ADD instruction in figure 5.8 takes seven microcycles. I split only the first instruction access. I could have split the other two, but it would cost two control words and the performance would not improve. I am doing well. There is only one state in the figure 5.8 sequence that is not hidden from the outside observer by an external bus cycle. Since the ADD instruction is important, I cannot let this happen. I will add hardware to make the instruction faster. I could save another microcycle if there was a way to get the next PC to the pads while the ALU is doing the ADD. Let's just add the hardware to do it.

The execution unit of figure 5.9 is the same as figure 5.6 except the internal A bus is now segmented into two pieces (called AP and AD). I can dynamically control the connection from each

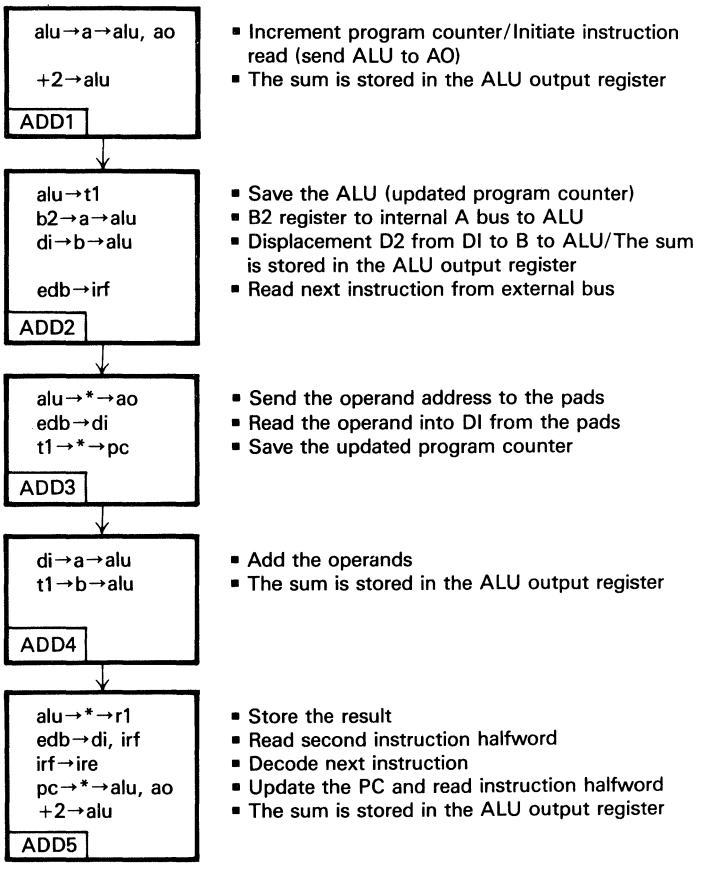

alu→a→alu, ao
+2→alu
ADD1
▪ Increment program counter/Initiate instruction read (send ALU to AO)
▪ The sum is stored in the ALU output register
alu→t1
b2→a→alu
di→b→alu
edb→irf
ADD2
▪ Save the ALU (updated program counter)
▪ B2 register to internal A bus to ALU
▪ Displacement D2 from DI to B to ALU/The sum is stored in the ALU output register
▪ Read next instruction from external bus
alu→*→ao
edb→di
t1→*→pc
ADD3
▪ Send the operand address to the pads
▪ Read the operand into DI from the pads
▪ Save the updated program counter
di→a→alu
t1→b→alu
ADD4
▪ Add the operands
▪ The sum is stored in the ALU output register
alu→*→r1
edb→di, irf
irf→ire
pc→*→alu, ao
+2→alu
ADD5
▪ Store the result
▪ Read second instruction halfword
▪ Decode next instruction
▪ Update the PC and read instruction halfword
▪ The sum is stored in the ALU output register

Figure 5.8 ADD instruction execution using split cycles

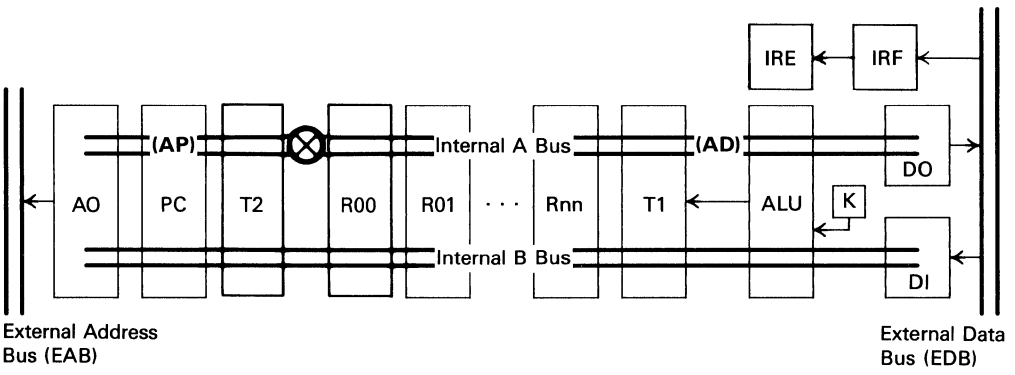


Figure 5.9 A more detailed execution unit with a bus coupler

control store word. Now I can eliminate one more microcycle from the ADD control word sequence (figure 5.10). I can't do any better as long as I am limited to a single external bus for instructions and data. This is almost like the hardware for a real commercial microprocessor execution unit. Mine is simpler because I want to show only how it operates. In practice, the execution unit will be more complicated because special hardware accommodates a greater variety of instructions, but the principles of timing and execution stay the same.

Global Chip Layout and Timing

Now it's time to back away from the execution unit a little to look at timing for the whole chip. If I look at what is occurring in the execution unit and on the external bus, I can deduce the timing for the rest of the chip. Figure 5.11 is a block diagram of a microprocessor. It shows the major communication paths for instructions, addresses, and data. Notice how the information comes in at the lower right corner and flows around the chip in a counterclockwise loop. All external addresses and data pass through the pads around the edge of the execution unit. Data into and out of the execution unit pass directly to and from the pads nearest the lower right corner of the chip. Instructions enter the execution unit on the data pads and are passed either to a temporary register in the execution unit (in the case of a displacement) or to the instruction decoder (in the case of an op code).

The output of the instruction decoders is the control store starting address for the control word sequence corresponding to the

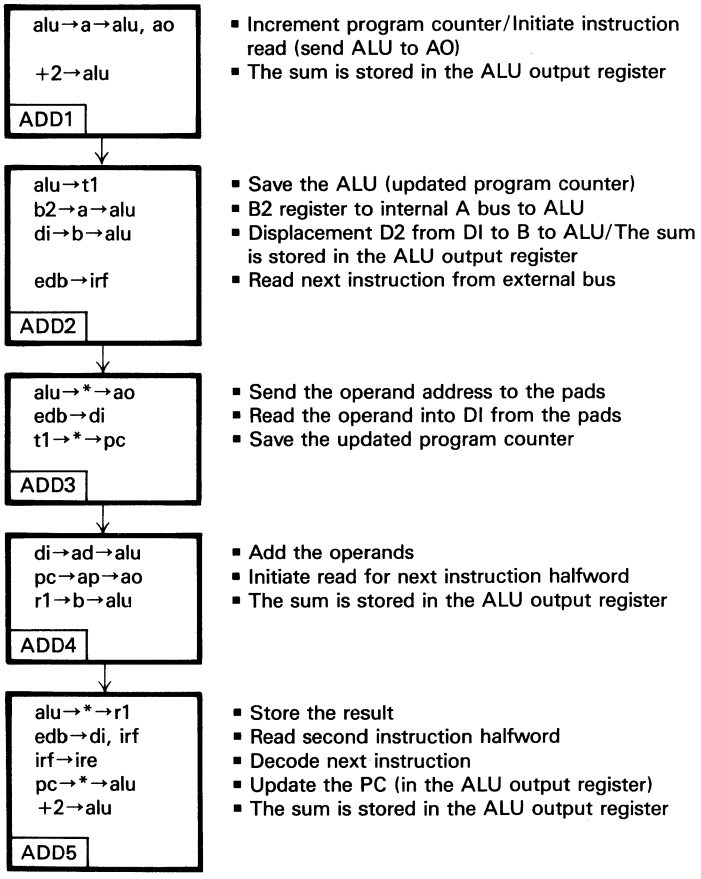

alu→a→alu, ao
+2→alu
ADD1
▪ Increment program counter/Initiate instruction read (send ALU to AO)
▪ The sum is stored in the ALU output register
alu→t1
b2→a→alu
di→b→alu
edb→irf
ADD2
▪ Save the ALU (updated program counter)
▪ B2 register to internal A bus to ALU
▪ Displacement D2 from DI to B to ALU/The sum is stored in the ALU output register
▪ Read next instruction from external bus
alu→*→ao
edb→di
t1→*→pc
ADD3
▪ Send the operand address to the pads
▪ Read the operand into DI from the pads
▪ Save the updated program counter
di→ad→alu
pc→ap→ao
r1→b→alu
ADD4
▪ Add the operands
▪ Initiate read for next instruction halfword
▪ The sum is stored in the ALU output register
alu→*→r1
edb→di, irf
irf→ire
pc→*→alu
+2→alu
ADD5
▪ Store the result
▪ Read second instruction halfword
▪ Decode next instruction
▪ Update the PC (in the ALU output register)
▪ The sum is stored in the ALU output register

Figure 5.10 ADD sequence with segmented internal bus

input op code. The address goes to the next state control and then to the control store. Part of the control word leaves the control store at the top, and part of it leaves at the bottom. At the bottom are the bits controlling the execution unit, and at the top are the bits for next state control. The control store bits leaving the bottom of the control store are decoded to control the execution unit.

The control store is used only for signals that change every state, but some of the information to do an instruction stays the same during the execution of the instruction. If the register pointers, for example, do not change during the instruction, there is no need to translate the register fields through the control store. The register pointers can be taken directly from the

IRE. The IRE is above the execution unit and directly feeds the control word decoders. Nothing on the chip is physically located very far from where it is used. The overall information flow on the chip is a loop. This loop is characteristic of the layout planning for a sensible design. It reduces wiring and delays. Since wiring tends to follow the flow of information, a loop means fewer wire crossings, too.

I already know some of the timing for the external buses and the execution unit, so let's figure out the rest of it. First I have

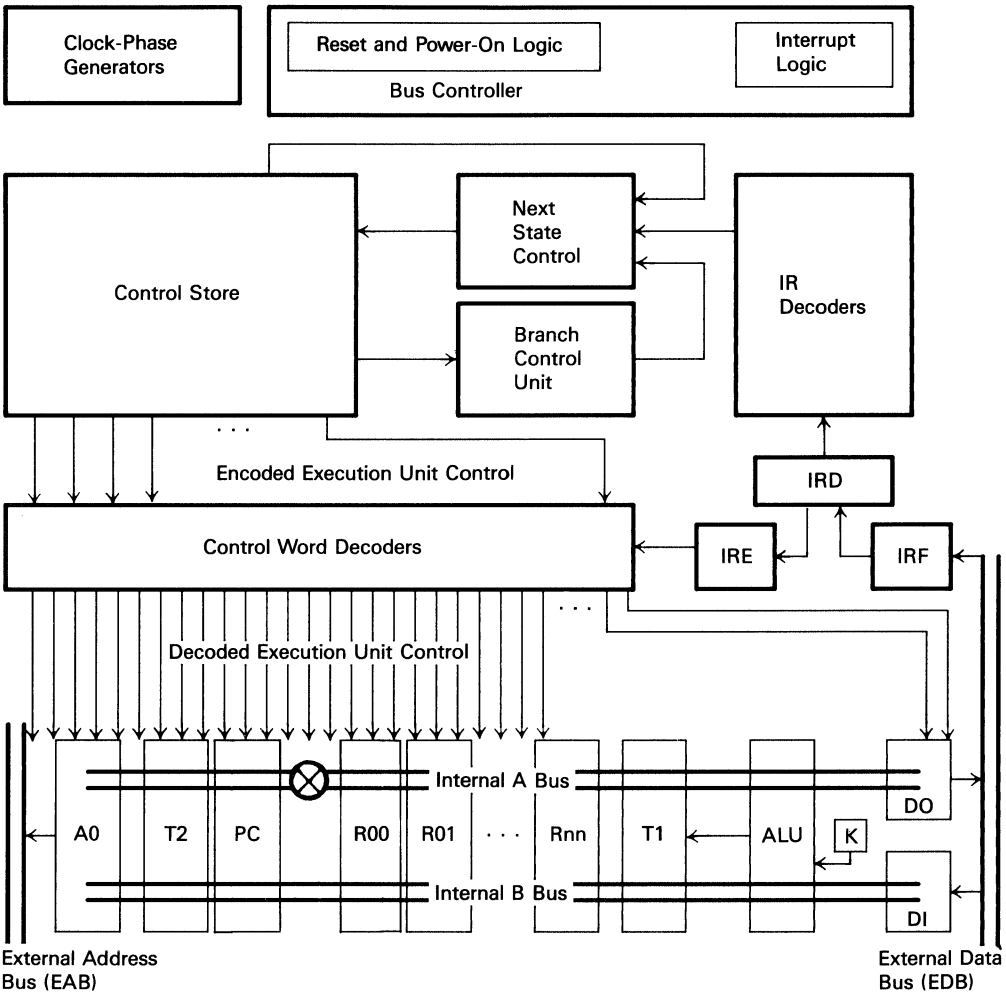


Figure 5.11 Microprocessor block diagram

to tell you how long the pieces take. The execution unit operates on a four-phase clock. A microcycle is one sequence of the four clock phases. The instruction decoders take one microcycle, and the control store takes half a microcycle. The control word decoders also take half a microcycle (see figure 5.12).

The execution unit operates in one microcycle and goes from phase 1 through phase 4. That means that the output of the control word decoders must be valid from the beginning of phase 1 through the end of phase 4, so I should latch it at phase 1. Since it takes two phases (half a microcycle) to operate, its input must be ready at phase 3 of the previous microcycle. I must latch the output of the control store at phase 3. Since the control store takes half a microcycle, its input address must be available at phase 1 of the previous microcycle. Since the IR decoders take one microcycle, their input must be available at

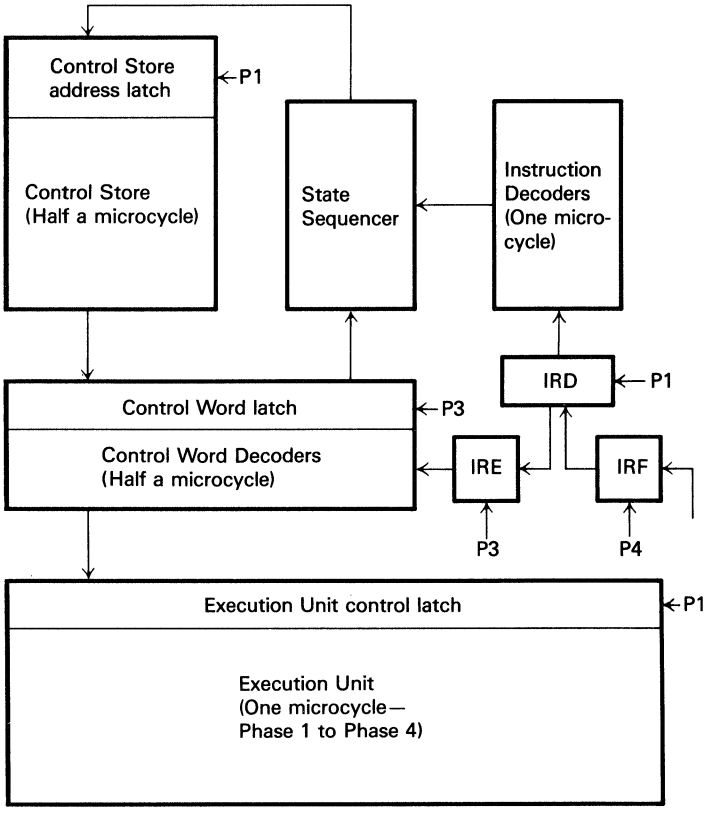


Figure 5.12 Microprocessor internal timing

phase 1—two microcycles before the execution unit is to operate. When I fetch an instruction, it is loaded into the IRF on phase 4. At the next phase 1, I could load the IRE and begin instruction decoding.

This presents more problems. Both the IRE and the control word are inputs to the control word decoders. The output of the control word decoders is saved at phase 1 (in a giant latch I call the execution unit control latch). Its inputs must be stable during the half microcycle before the end of phase 1. The output of the control store is okay because it is latched at phase 3. The IRE cannot change in phase 1, but it must be loaded at phase 1 to drive the IR decoders for a full microcycle. What do I do?

I had a problem with IRE anyway (see ADD5 in figure 5.10). In figure 5.11, I added the instruction register for decode (IRD) between IRF and IRE. IRD is there only to drive the instruction decoders, so it can be loaded anytime after IRF. It must be loaded at least two cycles before the end of the current control word sequence. (It can be loaded in the next-to-the-last state.) That allows one microcycle to decode the IRD value and one microcycle to retrieve the first control word and decode it. Figure 5.13 is the final ADD sequence.

Exceptions

I call the following "exceptions": interrupts, trace, bus error, reset, and power-on. There probably are others. Anything that isn't a valid instruction is an exception. This section tells what a microprocessor does with exceptions.

Interrupts

An interrupt is a request for a change in the normal instruction sequence. From the microprocessor's view, there are two kinds of interrupts. One kind comes from inside the chip (internal interrupts), and the other kind comes from outside the chip (external interrupts).

The following might cause an internal interrupt:

- fixed-point overflow
- divide-by-zero
- trace
- privilege exception (trying to do a supervisor instruction in user or problem state)

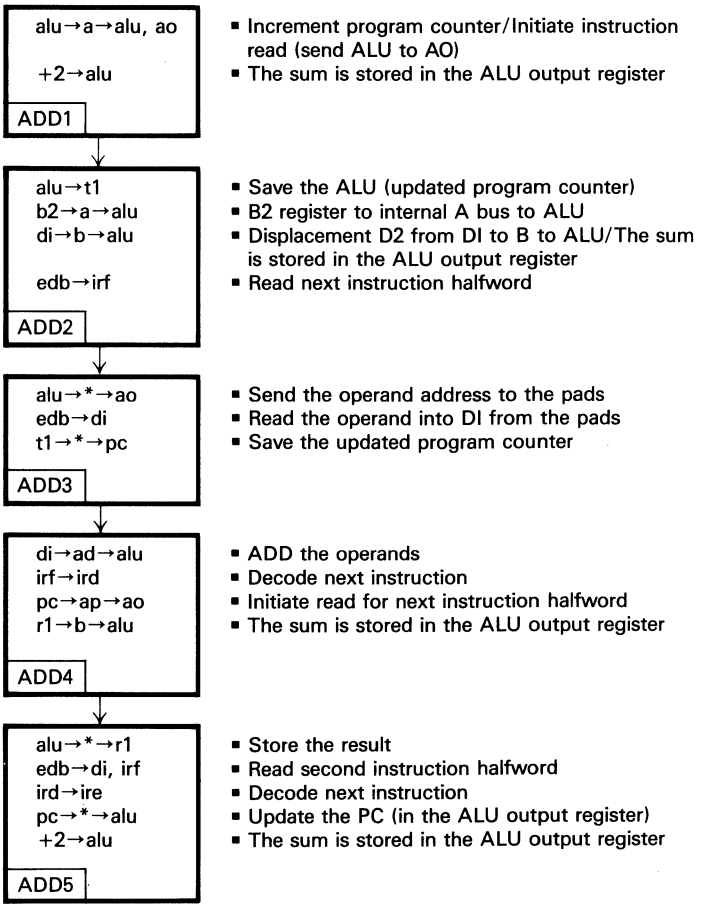

alu→a→alu, ao
+2→alu
ADD1
▪ Increment program counter/Initiate instruction read (send ALU to AO)
▪ The sum is stored in the ALU output register
alu→t1
b2→a→alu
di→b→alu
edb→irf
ADD2
▪ Save the ALU (updated program counter)
▪ B2 register to internal A bus to ALU
▪ Displacement D2 from DI to B to ALU/The sum is stored in the ALU output register
▪ Read next instruction halfword
alu→*→ao
edb→di
t1→*→pc
ADD3
▪ Send the operand address to the pads
▪ Read the operand into DI from the pads
▪ Save the updated program counter
di→ad→alu
irf→ird
pc→ap→ao
r1→b→alu
ADD4
▪ ADD the operands
▪ Decode next instruction
▪ Initiate read for next instruction halfword
▪ The sum is stored in the ALU output register
alu→*→r1
edb→di, irf
ird→ire
pc→*→alu
+2→alu
ADD5
▪ Store the result
▪ Read second instruction halfword
▪ Decode next instruction
▪ Update the PC (in the ALU output register)
▪ The sum is stored in the ALU output register

Figure 5.13 Final ADD sequence

- operation exception (trying to execute something that's not an instruction)

These conditions might cause an external interrupt:

- bus error
- peripheral (chip) service request
- reset request
- power-on
- test request

Because they are handled differently by the controller, I divide internal interrupts into two kinds: the kind that result from executing an instruction and the kind that do not. Arithmetic

overflow and divide-by-zero are internal interrupts that result from executing an instruction. Trace is an internal interrupt that does not result from executing an instruction.

Internal interrupts that result from executing an instruction are detected by the instruction's control word sequence. I do not need the hardware shown in figure 5.14 to deal with this but instead use a simple branch in the instruction's control word sequence. In this sense, these internal interrupts are handled immediately, so I call them immediate internal interrupts. An internal interrupt that does not result from executing an instruction is called a deferred internal interrupt. Trace is an example of a deferred internal interrupt. (Even though trace is on, instruction execution continues; recognition of the trace request is deferred until the current instruction completes execution.)

The bus controller synchronizes the external interrupt requests and sets bits to notify the controller of the presence of these requests. The control word sequence for the current instruction ignores these interrupt bits. At the end of the control word sequence, the interrupt bits cause the address of the interrupt control word sequence to be substituted for the address of the next instruction's control word sequence (figure 5.14). If the external interrupt can be masked, the mask (kept inside the microprocessor) is ANDed with the interrupt bit to select whether the interrupt control word sequence address goes to the control store address multiplexer.

It would be nice if all external interrupts were serviced only at instruction boundaries. That would make it easier to design the

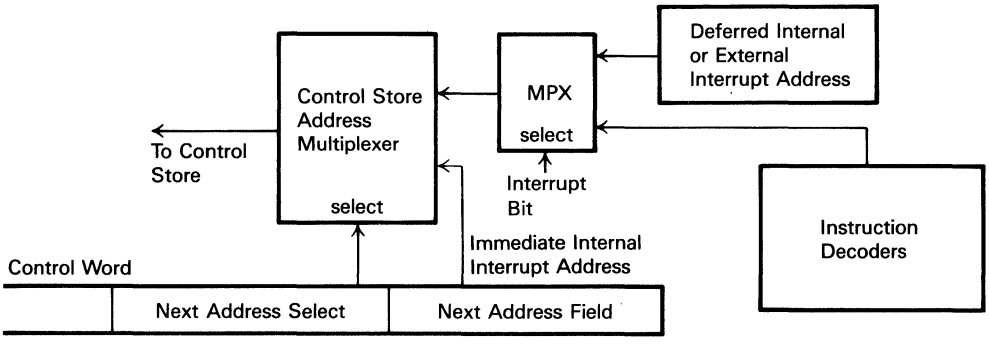


Figure 5.14 Control store address multiplexer for immediate internal, deferred internal, and external interrupts

controller and the interrupt control word sequence because the interrupts could be acknowledged when the internal state of the microprocessor was well defined. This works for what I call deferred external interrupts but not for bus errors, power-on (not much would be running with power-off), and some test and reset requests. These external interrupts I call immediate external interrupts.

Trace

To trace means to follow the trail of, to track down. A trace feature in a microprocessor offers a way to find out what a program is doing. A requester (program) turns on instruction tracing. This sets a trace bit in the microprocessor and causes a branch to the program being traced. The trace feature causes the traced program to be interrupted after one instruction, reports some internal state to the requester, and resets the trace bit. The requester analyzes the reported state (without tracing its own instructions) and turns on tracing for the next instruction in the program.

In the IBM Micro/370, trace is a System/370 program interruption with a special interruption code. The System/370 program interruption causes a program status word (PSW) swap and saves a 16-bit interruption code.

In the Motorola MC68000, trace does the following (after the current instruction completes execution):

1. Sets supervisor privilege state
2. Resets the trace bit
3. Pushes program counter and status register on supervisor stack
4. Executes instructions beginning at trace exception vector address

The requester gets a pointer to the instruction and some indication of what the instruction did (the status register). For most instructions, this is enough. For other instructions, the requester finds register contents or operand values in memory starting at the exception vector address.

The microprocessor treats trace like an ordinary interrupt. The control word sequence for the current instruction is not affected by the trace bit. At the end of the control word sequence for the current instruction, the last control word in the sequence at-

tempts to select the output of the instruction decoders as the next control store address (to go to the control word sequence for the next instruction). The trace bit causes the control store address to be substituted with the address of the trace control word sequence. See figure 5.15.

The control word sequence for trace will look like the control word sequence for any valid instruction. It will do the steps required by the specification for trace. The sequence of events in the trace control word sequence can be important. If the trace bit is part of the reported status, the trace bit cannot be reset before the status register is saved (either stored or temporarily saved). In the Motorola MC68000, reported status includes both the supervisor bit and the trace bit. The status register is saved (in a temporary register) before the trace and supervisor bits are changed.

Bus Error

Bus error is an immediate external interrupt. Bus error occurs when the external bus cannot complete an access request. Maybe you tried to read from a memory location that was not there. Bus error cannot wait for the end of the control word sequence because one of the control words is waiting for the completion of the external bus access. The bus error interrupt is a way to report a serious problem and to attempt to recover from the error. The bus error interrupt control word sequence is like the control word sequence for an ordinary instruction. The bus error interrupt control word sequence probably will store (in memory) the address of the failing instruction (PC) and the address on the pads when the bus error occurred.

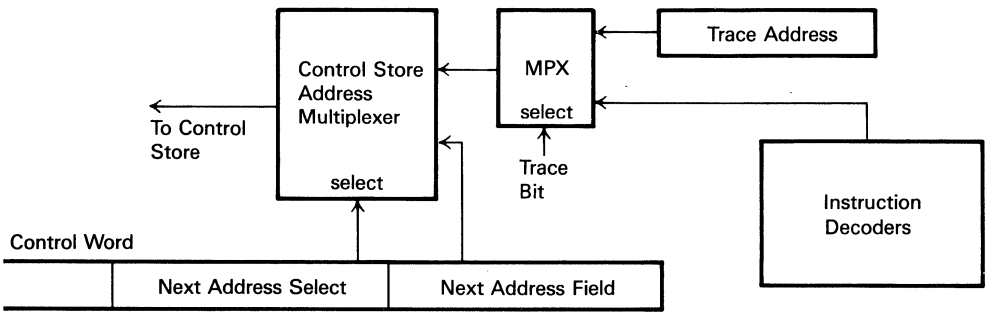


Figure 5.15 Control store address multiplexer for trace

I can't use the structure shown in figure 5.15 for immediate external interrupts such as bus error. These interrupts have to stop the controller during the current microcycle and change the control word sequence. I need the hardware shown in figure 5.16 for interrupts that can't wait until the end of the current control word sequence.

But we're still not there. Here's the problem: The execution unit is still running during the bus cycle that fails, and it will complete. If I am to substitute a new control word at the next microcycle, I have to inhibit the next microcycle in the execution unit. Here's what is happening: The execution unit puts an address on the pads, and the bus controller starts a bus cycle. The execution unit completes a microcycle. The control store completes its overlapped access to the next control word. The bus cycle fails. I now want to present a new address to the control store, but there is already a new control word at the output of the control store waiting to be used by the execution unit. I can't let this control word execute because I need to say where the bus cycle failed. It may have failed on the last control word in the instruction's control word sequence. If I let the next control word execute in the execution unit while I access the first control word of the bus error control word sequence, I could lose state information (such as the instruction length count or the program counter pointing to the failed instruction) associated with the bus error.

Reset

The request for a reset is synchronized by the bus controller. A reset is just like a bus error interrupt. I do not want to wait for the end of the current control word sequence. (I might be doing the reset because the hardware is stuck in an illegal state.) I want to stop doing the current instruction immediately and initialize the microprocessor. Reset differs from the bus error interrupt in that I do not have to preserve the internal state of the execution unit. I am going to set the internal state of the execution unit without reference to the current state. The reset interrupt control word sequence resets the status register and introduces a new program counter. It uses the same hardware as the immediate external interrupts (figure 5.16).

Power-On

Special circuits start the microprocessor after the power switch is turned on. The controller must come up in a known state;

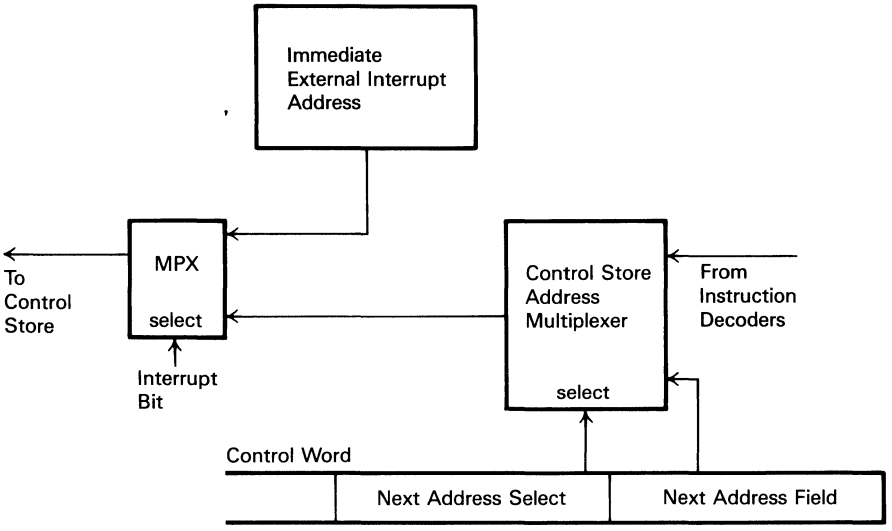


Figure 5.16 Control store address multiplexer for immediate external interrupts

then it should run the power-on control word sequence to start the microprocessor. The power-on control word sequence probably will initialize the status register and set the PC. It will have its own clocking and will use the same hardware as the immediate external interrupts.

Immediate external interrupts, reset, and power-on are not subject to masking by the programmer. These interrupts occur as soon as they are recognized—that is, immediately.

Summary of Exceptions

I have shown how a microprocessor handles exceptions and the required hardware. Figure 5.17 shows the control store address multiplexing with all the interrupt address sources.

I have named four categories of interrupts: immediate and deferred, internal and external. (These have no fixed relation to interrupts defined in the architecture document. They are how a microprocessor designer views interrupts.) Interrupts are defined by where they come from (inside or outside the chip) and by how the designer deals with the interrupts (now [immediate] or later [deferred]). Internal (for instance, arithmetic) interrupts use simple branches in the control word sequence. Deferred internal

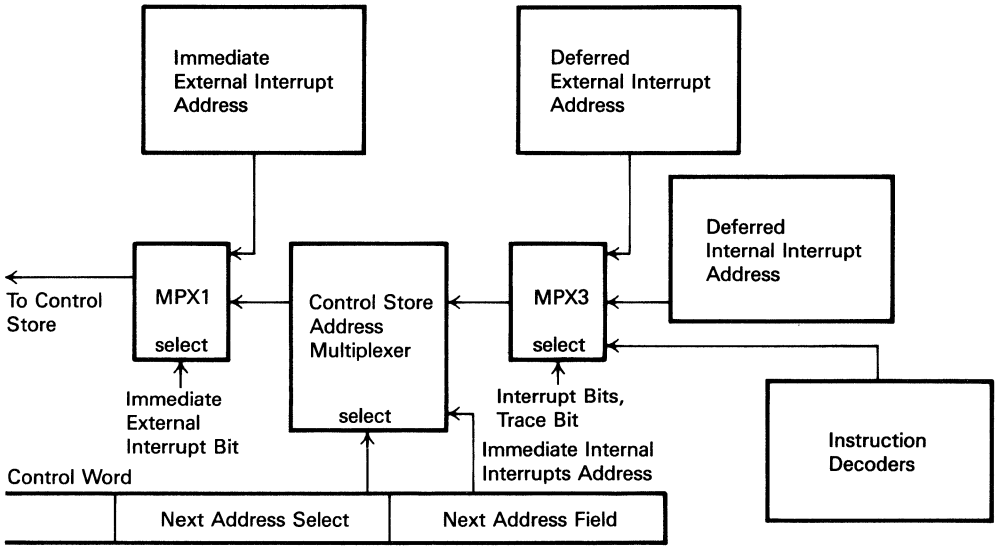


Figure 5.17 Control store address multiplexer for exceptions

and external interrupts substitute the address of the interrupt control word sequence in place of the pointer to the next instruction's control word sequence. Deferred internal and external interrupts are taken only at instruction boundaries. Immediate external interrupts substitute the address of the interrupt control word sequence for the address of the next control word and suspend action in the execution unit (while a new control word is accessed).

Previously, I defined an interrupt as a change in the normal order of executing instructions. But a designer does not necessarily care about whether instructions are executed out of the normal order. A designer cares about a change in the normal order of executing control words. For interrupts handled by a conditional branch within the control word sequence, there is really no change in the normal order of executing control words. The control word sequence just takes path A or path B. To a designer, immediate internal interrupts are not, in a sense, really interrupts. Deferred internal and deferred external interrupts are real to a designer because the normal order of executing control words is interrupted between control word sequences on a sequence boundary. Immediate external interrupts are real, too,

because the normal order of executing control words is interrupted within a control word sequence on a control word boundary.

Multiple Interrupts

What if more than one interrupt is present at one time? Let's suppose I am executing a Divide instruction with a zero divisor. The trace bit is set. A direct memory access (DMA) peripheral wants service (it is presenting an external interrupt request), and the operand access cannot complete. The winner of the contest is determined by the architect; the user's manual should rank the interrupts. Here is a typical ranking:

1. Immediate external interrupts
2. Immediate internal interrupts
3. Deferred internal interrupts
4. Deferred external interrupts

If the divide control word sequence cannot get the operands, it will not know there is a divide fault. The bus error interrupt generated by the failing operand access will outrank the arithmetic interrupt. In the absence of an immediate external interrupt, the arithmetic interrupt will rank highest because it is handled within the control word sequence. (The other interrupts have no indication that the instruction is not proceeding normally.) The internal interrupt (trace) ranks higher than the deferred external interrupt by controlling the hardware selection of the multiplexer (MPX3 in figure 5.17).

What if there are multiple interrupts of each kind? There is no problem with the immediate internal (arithmetic) interrupts because only one can occur per instruction. But there can be several peripheral chips that all want attention (the printer, the disk, the keyboard, the modem, and so on). There can be many deferred external interrupts, deferred internal interrupts, and immediate external interrupts. The interrupts in each category must be sorted out by hardware.

External interrupts are ranked by external hardware if there is only one pin for external interrupt requests. If there are several pins, as there are on Micro/370, the external interrupt requests must be ranked on the chip. All interrupts within a single category must be ranked. I use a PLA for each category to rank the

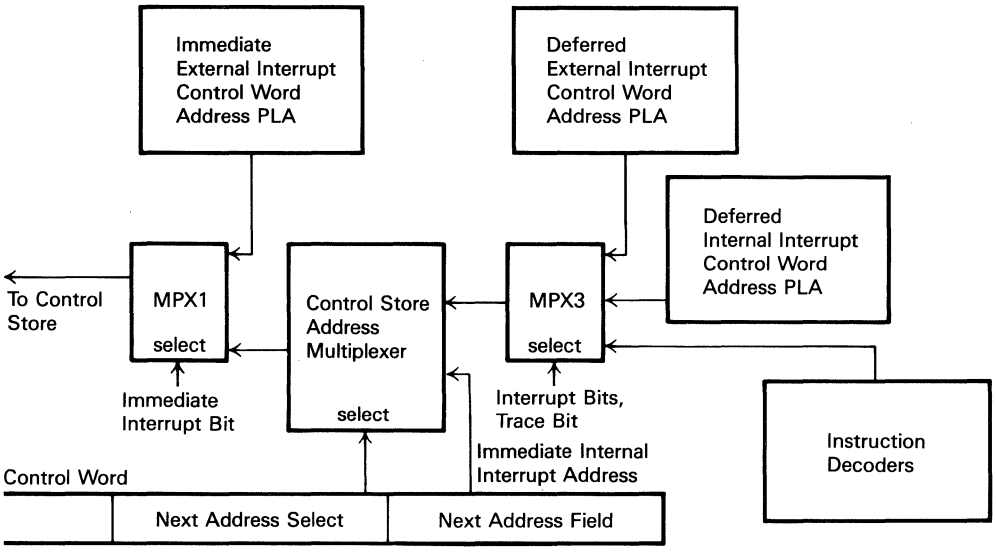


Figure 5.18 Control store address multiplexer hardware for exceptions

requests and to produce the address of the winning interrupt control word sequence. Figure 5.18 shows the hardware needed for the control store address multiplexer.

Control Store Address Selection

I am almost through with the control store address multiplexer. I need two more things: control store branching and additional addressing from the instruction decoders.

Branches

I have to have control word branches. Some of the instructions need them, and I need them to implement algorithms in control word sequences. I won't get a hardware divide unit on the chip this year, so I will have to implement divide with some kind of (looping) algorithm.

Figure 5.19 is the control word format. The field labeled "Next address selection" selects the next control store address for normal operation. (In the event of an exception, the highest ranking exception condition determines the next control store address source.) So far, I have shown only two sources for the

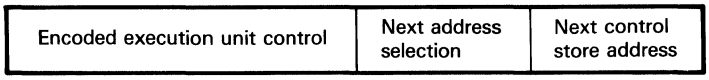


Figure 5.19 Control word format

(normal) control store address. One is the next control store address field from the control word. The second source is the instruction decoders. The third source will be branch addresses.

Branch addresses are implemented using part of the next control store address field and a few (1 to 3) bits from the branch control unit (see figure 5.11). Assume that I am allowed four-way branches in the control store. The branch unit produces 2 bits. These 2 bits take the place of 2 bits from the control store next address field in the control word (figure 5.20).

Instruction Decoding

Until now I have shown one address for instruction decoding, but you can use more. The Motorola MC68000 has about fourteen address modes. Most of the address modes can be used with any instruction. The control word sequences for calculating "effective address" are shared among instructions. A micro-

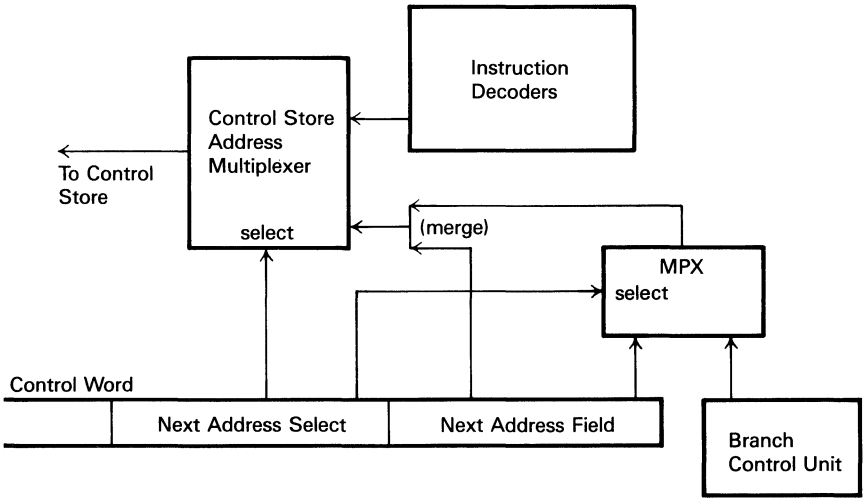


Figure 5.20 Control store address multiplexer for branches

processor can use at least two instruction decoders. (Call them A1 and A2.) One of the instruction decoders points to the address calculation control word sequence and the other points to the instruction control word sequence.

Micro/370 uses two instruction decoders. If there is no address calculation control word sequence, the first instruction decoder points to the instruction control word sequence, and the second instruction decoder is not used.

The Motorola MC68000 uses three instruction decoders, called A1, A2, and A3. There are three instruction decoding cases. Figure 5.21 shows which decoder points to which control word sequence for the three cases. (I lied a little. Physically, the Motorola MC68000 has only two instruction decoders, designated A1 and A2/A3. The A2 and A3 decoders are physically implemented by one PLA, which has one AND and two OR arrays.)

Operation, Specification, and Privilege Exceptions

I can put any bit pattern into the instruction decoders, but not all of them are legal op codes. The ones that are not legal cause an operation exception, which can be detected in the instruction decoder PLAs or with separate hardware.

A specification exception occurs when the op code is legal but something else is wrong with the instruction. In System/370, some instructions must designate an even-numbered register (they use an even-odd register pair for operands). If the programmer uses a multiply instruction with an odd-numbered register designated in the R1 field, a specification exception will occur.

	Control Word Sequence		
	Immediate Data or Address Calculation	Address Calculation	Instruction
Case 1			A1
Case 2		A1	A2
Case 3	A1	A2	A3

Figure 5.21 Instruction decoder operation

The instruction decoder points to the multiply control word sequence if the register designated by R1 is even. It points to the specification exception control word sequence if the register designated by R1 is odd.

Privilege exceptions can be implemented as deferred internal interrupts. If some instructions can be executed only in a supervisor state, the instruction decoders cough out the address of the instruction control word sequence if in supervisor state and the address of an exception sequence if not. If the supervisor state bit is included in the instruction decoding, you do not need a separate PLA to check for supervisor state.

Figure 5.22 is the control store address multiplexer for all control store address sources.

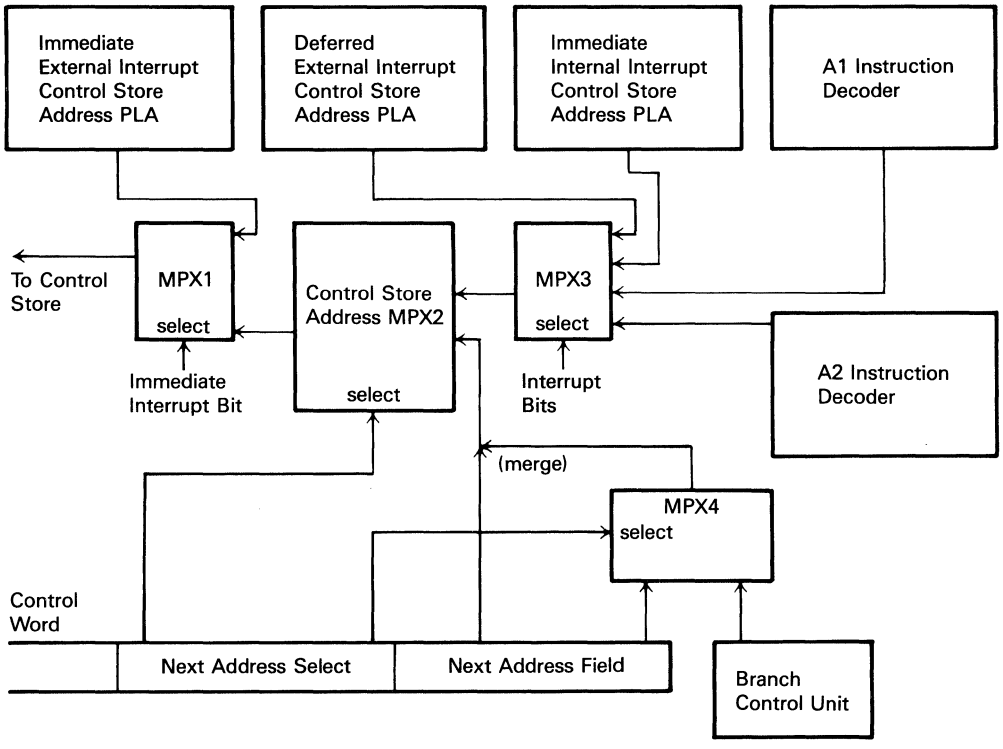


Figure 5.22 Control store address multiplexer for branches

Control Store

Operation

The address to the control store (the output of MPX1 in figure 5.22) is latched at phase 1. The control store takes two clock phases to operate. The output of the control store is latched at phase 3. The address enters the control store at the top and is decoded through the center of the control store. Both the next address select field and the next address field of the control word leave the control store at the top. There they go to the control store address multiplexer. They come out the top because most of the addresses for the control store are available at the top of the chip. There is more room there, and the lines do not interfere with the control lines to the execution unit. The control bits for the execution unit leave the control store at the bottom. They go directly into the control word decoders.

Organization

The control store is a dense ROM. The control store is split into upper and lower physical sections (levels). The upper level contains the next address select fields and the next address fields of the control word. These fields make up microwords. The lower level contains the execution unit control bits. These fields make up nanowords. The control store also is split left and right. The address decoder runs down the middle because its longest signals only have to go half as far. The overall width (sum of left and right sides) of the control store is set by the size of the execution unit and the instruction decoders. It is wider than one control word. In the Motorola MC68000, there are sixteen microwords in the top rows of the control store and four nanowords in the bottom rows (assuming top rows and bottom rows are the same length). This means the nanowords are about four times as wide as the microwords. In the IBM Micro/370, the microwords are 18 bits, and the nanowords are 71 bits with 1 spare. In the Motorola MC68000, the microwords are 17 bits, and the nanowords are 66 bits with 2 spares ($4 \times 17 = 68$).

Figure 5.23 shows the physical organization of the two-level control store. In the IBM Micro/370, the microword store is above and to the left of the nanoword store. It sits on its side, with inputs and outputs on the right.

In the Motorola MC68000, there is a 4-to-1 multiplexer at the bottom of the control store and a 16-to-1 multiplexer at the top.

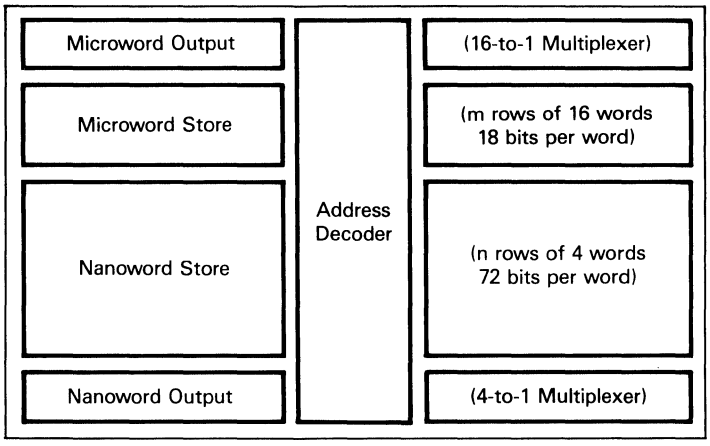


Figure 5.23 Two-level control store (organization)

In Micro/370, both control stores use a 4-to-1 multiplexer at the output. The bits of the words in each control store row are interleaved, so that when they exit the multiplexer, they are right next to where they are used. In Micro/370, the two branch bits are used to control the multiplexer selection because the multiplexer select bits are the last needed and the bits coming from the branch unit are the last available. The bits from the branch unit are on the critical path for the control unit. (Note: This configuration constrains placement of words in the control store.)

In both Micro/370 and the MC68000, the control store is implemented with a dynamic ROM. That allows two rows to share a ground line (normally, each row has its own ground line), which makes the circuits smaller. The ROM can be dynamic because the output is always latched once each microcycle. The output does not have to be static. By dynamic, I mean that for a fixed input, you see the output go to a value for a while and then go away. You have to latch it while it is valid. A static ROM maintains its output as long as the input remains unchanged (after some initial delay).

Nanoword Decoder

Nanoword Encoding

The nanoword is divided into fields. Each field controls one to a few resources in the execution unit. The nanoword fields are in the same physical order as the pieces of the execution unit. I

assign bits to each nanoword field according to how much it controls. Generally, I assign a separate nanoword field for each independent resource in the execution unit. If some pieces of the execution unit are never (or almost never) used at the same time, I might make them share a control field—but only if the controlled pieces are next to each other or if I can move them next to each other. If the pieces cannot be next to each other, sharing the nanoword field is not worth the effort because the decoded control lines would have to cross.

I would assign one nanoword field to the PC register, for example. I would then enumerate control events for the PC. The PC probably would drive the A bus or the B bus, or you could write to it from either the A or B bus. That's four control events. (If combinations of these control events occur in the same flowchart state [such as the PC driving the A bus and the B bus, for example], they become separate control events and you need more bits.) It looks as if I need 2 bits, but I really need 3 because I have to allow for the case that does nothing.

Nanoword Decoding

The nanoword decoder is shown in figure 5.24. Its inputs are the nanoword from the control store and the bits of IRE. Here is what the nanoword decoder does:

- decodes control store nanoword bit fields
- combines control signals of the nanoword fields and the IRE fields
- introduces timing signals to the logic terms
- geometrically aligns control signals with corresponding execution unit control points

Static Decoding

I call the decoders driven by the instruction register static decoders, because they do not change each microcycle. The fields of an instruction do not change during the execution of that instruction, but what you do because of the fields in the instruction does change from microcycle to microcycle. That's the stuff in the control store, and it must be decoded every microcycle. Static decoders make the instruction a fixed extension of every control word in that instruction's control word sequence. Op code, register fields, length fields, masks, and some ALU op codes are handled by static decoders.

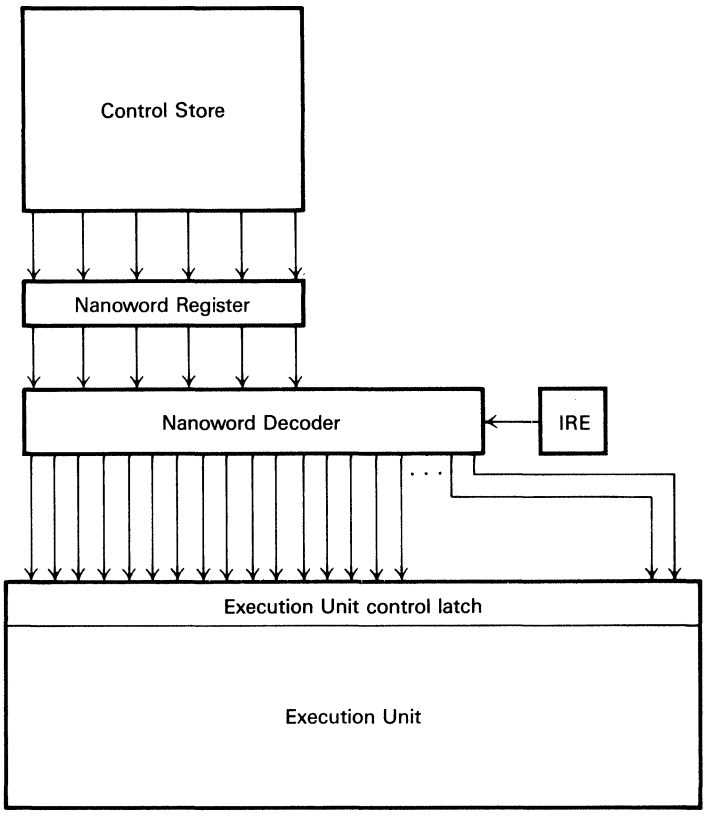


Figure 5.24 Microprocessor nanoword decoding

Timing

Timing is introduced to the execution unit control in the nanoword decoder. In the execution unit, source and destination transfers happen during clock phases 1 and 3, respectively. Since the nanoword is latched at phase 3, the source control signals can pass right through the execution unit control latches from the nanoword register. (Source control signals do not have to be latched.) The destination control signals must be latched at phase 1 to prevent hazards, because the nanoword is latched in the same clock phase (3) in which the destination control signals are driving the execution unit. Figure 5.25 shows how this works. Timing signals are uniformly introduced in the nanoword decoders, which reduces the chances for clock skew or hazards to cause logic errors.

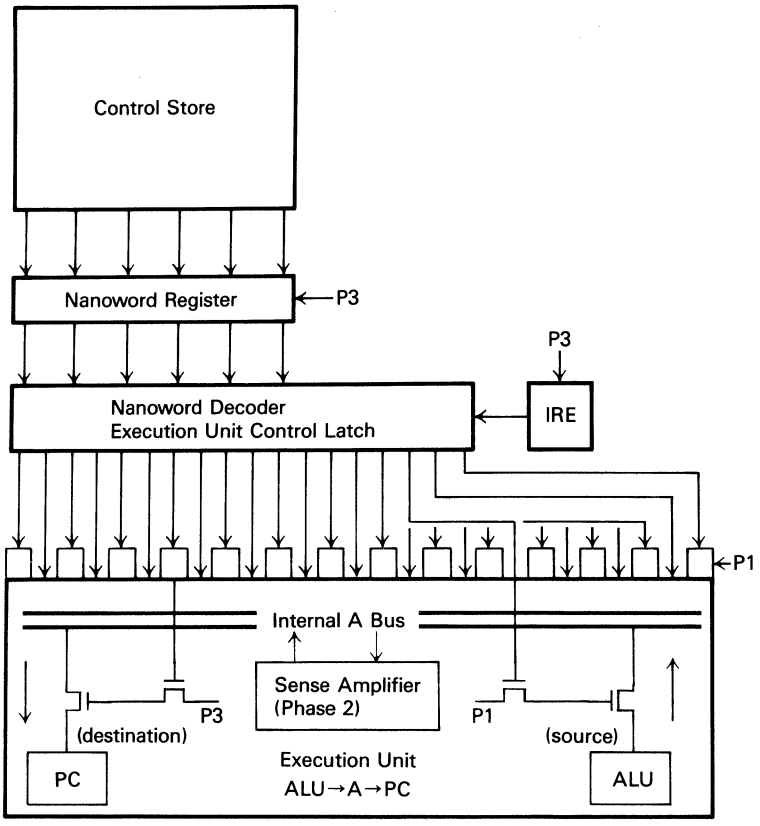


Figure 5.25 Microprocessor nanoword decoder, latch, and control

Decoding

Register Control An instruction such as Multiply Register takes several microcycles. Execution unit operations change with each microcycle, but the register pointers stay the same. There is no need to use a field in the nanoword to point to the register. I can let the register fields in the instruction select the registers and use a nanoword field to tell when and how I am using the selected registers.

As with everything else so far, it isn't that simple. The Multiply Register instruction in System/370 is an example. It specifies a register pair for the result, not just a source and a destination register. You have to check the instruction to be sure the R1 field specifies an even register. If it does not, you do not exe-

cute the instruction. If it does, you have to get the multiplicand from the odd register of the even-odd pair, multiply it by the contents of the register specified by the R2 field of the instruction, and put the result in the register pair specified by the R1 field. You can't just use the register pointers in the instruction register; you have to be able to force the low-order bit of the register pointer to be a one sometimes.

The architects of System/370 went even further, as some instructions have a base register field before the displacement. (There are two base-displacement fields for storage-to-storage instructions.) Worse, if the base register field (called B1 or B2) contains a zero, you do not use register zero but instead use the value zero. That is also true for the R2 field *sometimes,* which means that you have to look at the R1 field, the R2 field, and wherever the B1 and B2 fields are. Plus you have to be able to substitute a zero for register zero in some cases. And sometimes you have to force the low-order bit of the register pointer to be a one.

That isn't all. The System/370 architects defined instructions that write their results into specific registers. For example, the Translate and Test instruction (TRT) puts a byte in general register 2 and an address in general register 1. There is no field in TRT indicating general registers 1 and 2. There is an operand length field where register fields usually sit. Your register control hardware must be able to figure this out and point to registers directly.

The System/370 Load Multiple (LM) and Store Multiple (STM) instructions load or store (respectively) from one to sixteen general registers. The instructions only specify a beginning and ending register number. You have to load or store the registers in sequence beginning with the register specified by the R1 field and continuing (modulo 16) through the register specified by the R2 field. This means that you need hardware to generate sequential register pointers and to decide when to quit. Hardware to do this is shown in figure 5.26. You need one of these register control units for the A bus and another for the B bus. Even that isn't all there is to register control. Timing signals are introduced after the 5-to-17 decoder.

System/370 specifies a set of sixteen 32-bit general registers. In Micro/370, there is another set of sixteen 32-bit registers called

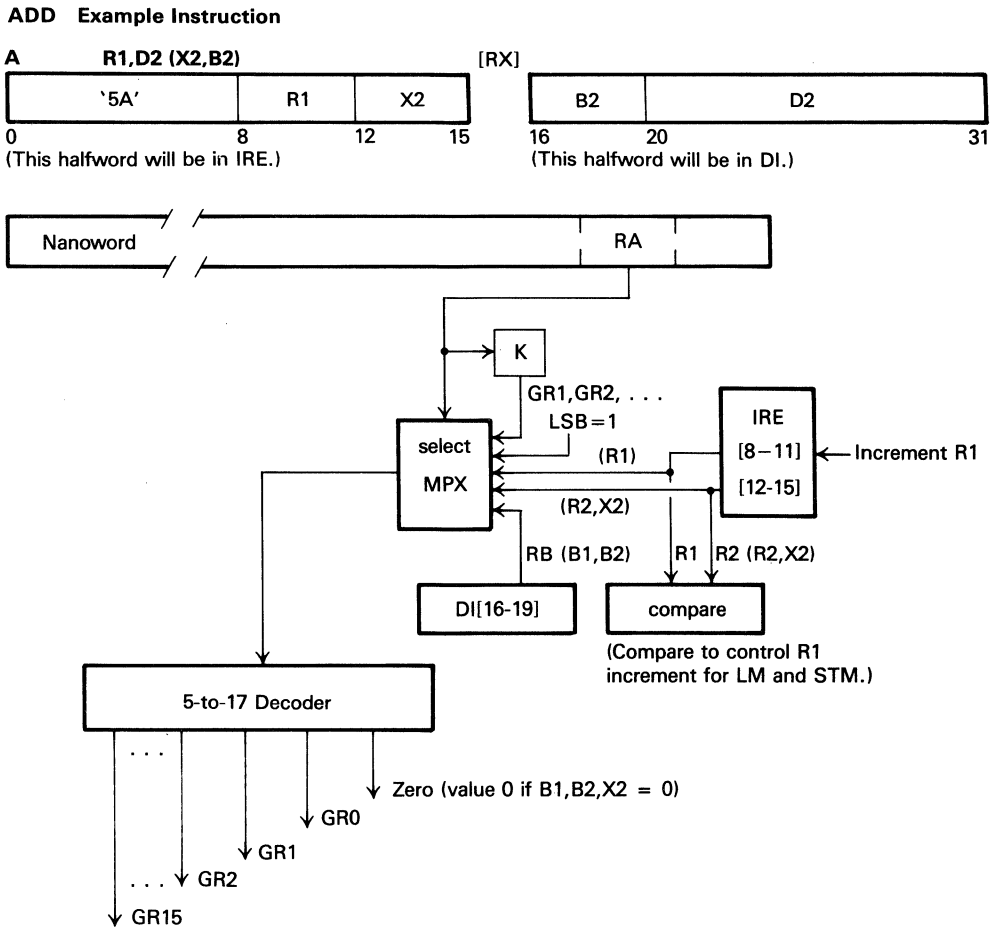


Figure 5.26 Register control decoder

shadow registers. A bit controls whether reading is from the general registers or from the shadow registers. Another bit controls whether writing is to the general registers or to both sets of registers. These signals are introduced after the 5-to-17 decoder.

ALU Function Control It appears that instructions such as OR Register can use the op code to select the ALU operation directly. This would save having to specify the ALU operation in the nanoword field. It might let you share the control word sequences for several instructions. (The control word sequences

for AND, ADD, OR, XOR, and SUB might differ by no more than the ALU operation.) But some instructions need more than one op code. Multiply, for example. I can't just have the ALU do a multiply (yet). I have to use adds and subtracts. Even simple instructions such as OR might have address calculations that need ALU operations other than one implied by the op code.

I do not want to give up the possibility of sharing similar control word sequences, but I need to be able to use several ALU operations in a single control word sequence. How can I do that? I use a table of ALU functions. The instruction register selects the row in the table, and the nanoword selects the column. Figure 5.27 is the ALU and condition code control table from Micro/370. (I will discuss condition codes shortly.)

As I wrote the flowchart for each System/370 instruction, I assigned it to a row in the ALU and condition code control table. The row in the table shows which ALU operations are used and which System/370 condition codes should be set. When I assigned an ALU operation, I had no idea how the System/370 condition code setting would be achieved, only how it must be assigned. I gave each System/370 condition code setting a letter of the alphabet. In a separate place, I wrote the equation corresponding to the letter for condition code setting (see figure 5.27). When the flowcharts were complete, I looked at the condition code equations I used and put them in a Karnaugh map. I tried to place them to minimize the PLA for implementation.

Several ALU operations are common to all instructions because you can never count on any instruction finishing normally. Some (instructions) contain illegal data. Some have forbidden formats. Some cause overflows. Sometimes the memory address is wrong or the memory system is broken. Lots of unusual things can happen. In System/370, for example, the architecture says specific things about what to do in each of these cases. The microprocessor cannot just quit. It has to try to recover. It stores the old state (PSW) and perhaps some information about what happened, reads a new PSW, checks it for validity, loads it, and begins running instructions at the location it specifies. All this has to be done potentially for any instruction in the IRE.

The common ADD and SUB columns are used by many instructions for address calculation. The AND and XOR columns are used by the special recovery and exception control word sequences for checking the format of the new PSW.

Column (Selected by Nanoword) 1		2		3		4		5		6		System/370 Instructions
A+B	A	A+B	7	A+B	6	A−B	A	AND	A	XOR	A	A,AH,AR,LTR,LNR, SRDA,SRA,SLA, SLDA
A+B	A	A+B	D	ADDC	E	A−B	A	AND	A	XOR	A	AL,ALR,DIAG
A+B	A			A−B	A	A−B	A	AND	A	XOR	A	BXH
A+B	A			B−A	A	A−B	A	AND	A	XOR	A	BXLE
A+B	A	A−B	1			A−B	A	AND	A	XOR	A	C,CR
A+B	A			A−B	C	A−B	A	AND	A	XOR	A	CDS,CS
A+B	A	A−B	9	OR	A	A−B	A	AND	A	XOR	A	CL,CLC,CLCL,CLI, CLM,CLR
A+B	A	ADDC	A			A−B	A	AND	A	XOR	A	CVD
A+B	A	B−A	A	OR	A	A−B	A	AND	A	XOR	A	D,DR,EX,M, MH,MR
A+B	A	A−B	7	ADDC	E	A−B	A	AND	A	XOR	A	DIAG,LCR,LPR,S, SH,SR
A+B	A	A−B	D			A−B	A	AND	A	XOR	A	DIAG,SL,SLR
A+B	A	A+B	5			A−B	A	AND	A	XOR	A	ICM
A+B	A	A−B	5			A−B	A	AND	A	XOR	A	MVCL
A+B	A	AND	C	OR	A	A−B	A	AND	A	XOR	A	N,NC,NI,NR
A+B	A	OR	C			A−B	A	AND	A	XOR	A	O,OC,OI,OR
A+B	A	AND	A	OR	A	A−B	A	AND	A	XOR	A	STNSM
A+B	A	OR	A	OR	A	A−B	A	AND	A	XOR	A	STOSM
A+B	A	AND	C	XOR	B	A−B	A	AND	A	XOR	A	TM
A+B	A	A−B	9	XOR	8	A−B	A	AND	A	XOR	A	TRT
A+B	A	A+B	4			A−B	A	AND	A	XOR	A	TS
A+B	A	XOR	C			A−B	A	AND	A	XOR	A	X,XC,XI,XR

1. "A+B 7" means ALU operation is ADD, System/370 condition code is created by box 7 in the Karnaugh map below.
2. ADDC is A+B+1.
3. Unlisted instructions do not use column 2 or 3 operations.
4. DIAGnose appears on several lines—different forms.

Karnaugh Map (How System/370 condition codes come from the standard set of condition codes produced by the ALU)

cc0 cc1	0	1 $\overline{N}\cdot\overline{V}\cdot\overline{Z}+N\cdot V$ $N\cdot\overline{V}+\overline{N}\cdot V$	3	2
cc0 cc1	4 0 N	5 $\overline{N}\cdot\overline{V}\cdot\overline{Z}$ N	7 $\overline{N}\cdot\overline{Z}+V$ N	6 $cc0\cdot\overline{N}+$ $cc1\cdot\overline{N}+\overline{N}\cdot\overline{Z}$ N
cc0 cc1	C 0 $\overline{Z}$	D C $\overline{Z}$	F 0 $\overline{Z}$	E cc0 $\overline{Z}$
cc0 cc1	8 $\overline{Z}$ 0	9 $C\cdot\overline{Z}$ $\overline{C}$	B Z·cc1 cc1	A cc0 cc1

(An empty box is a "don't care" for both bits.)

(cc0 keeps its old value in boxes A and E.)

Figure 5.27 Micro/370 ALU and condition code control table

Condition Codes Not all instructions set condition codes. In the Motorola MC68000, for example, Move instructions and Address Manipulation instructions do not affect condition codes. Logical instructions such as AND, OR, and XOR affect the condition codes differently than arithmetic instructions such as Add and Subtract. System/370 goes further. It has Add Logical and Subtract Logical. These instructions produce the same results as Add and Subtract but have different condition codes. How do I handle this in the design?

ALU operations produce a standard set of condition codes. These are:

- Z zero bit (1 means zero result)
- V overflow bit (1 means overflow)
- N sign bit (1 means negative)
- C carry bit (1 means carry)

In Micro/370, these condition codes are saved after each ALU operation for both the word result and the low-order byte result. System/370 architecture defines 2 condition code bits, which bear no apparent relationship to the sensible condition codes in the standard set. The architectural condition codes are fabricated through a logic network using the standard condition codes, the instruction register, and the previous condition code bits (from the PSW).

Some condition codes in System/370 are not the result of arithmetic operations. These condition code bits cannot be produced by the logic network. One example is the Move Character Long (MVCL) instruction, which sets the condition codes to indicate relative operand lengths—unless the operands overlap destructively. If there is destructive overlap, the condition code is supposed to be set to 3. Microcode detects destructive overlap cases using a variety of tests. If there is destructive overlap, I load a 3 directly into the PSW condition code bits using a constant from the execution unit. Condition codes that cannot be derived from the standard set are loaded directly using constants from the execution unit.

Figure 5.28 shows the logic for setting the System/370 condition codes in Micro/370. The op code from the IRE says whether the operation is a byte or word and how the condition codes are officially defined. An ALU and condition code control field (ACC) from the nanoword helps IRE control the ALU and condition codes. IRE gives static control, and ACC gives dynamic control.

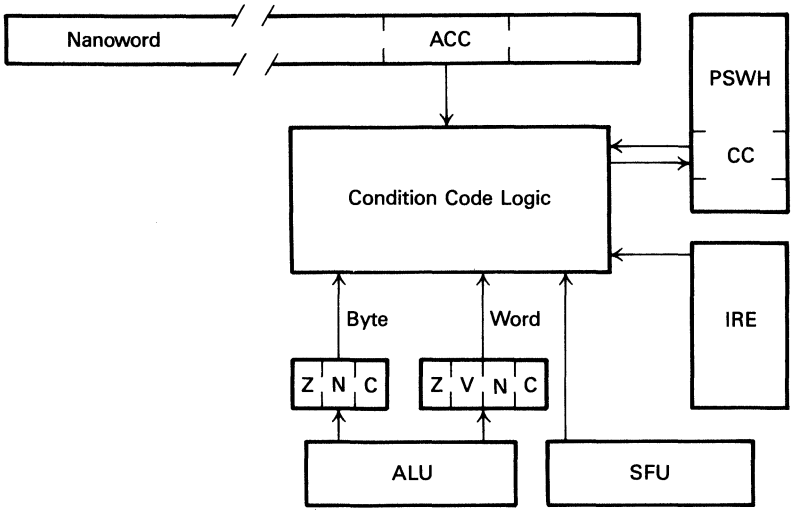


Figure 5.28 Micro/370 condition code control logic

Communication between Execution Unit and State Sequencer

I have already said that the state sequencer controls the execution unit. Now I will describe how the state sequencer knows what is going on in the execution unit.

Status Register

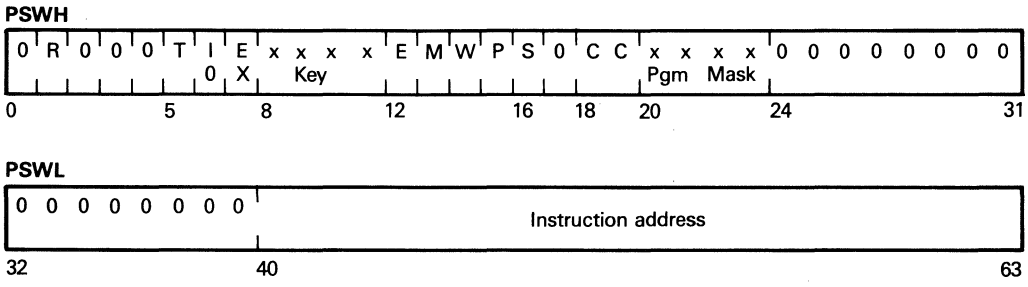
The Status Register (SR) contains program-controlled state information. Usually this means condition code bits, interrupt masks, the PC (may be considered separately), and mode bits. In System/370, the SR is called the program status word (PSW), which actually is two words. Its format is shown in figure 5.29.

The Motorola MC68000 has the standard set of condition codes plus a special carry bit (x) for doing extended precision arithmetic. In System/370, the condition codes are 2 bits derived from the standard condition codes (and a few other things). Condition code bits are set implicitly by the programmer (as a side effect of executing an instruction defined to affect the condition codes). Micro/370 and the MC68000 have instructions that affect the condition codes explicitly.

The interrupt mask bits say which interrupts are allowed. In the Motorola MC68000, the interrupt mask is a level number. All in-

terrupts are ranked. Interrupt requests with a level less than or equal to the current interrupt level are ignored. In System/370, one bit controls each interruption type. There is a bit for I/O interruptions, a bit for external interruptions, a bit for machine-check interruptions, and 4 bits for various program interruptions (see figure 5.29). In the MC68000, the interrupt mask is set to the level of the recognized interrupt. This prevents lower priority or equal priority interrupts from interfering with the interrupt service routine. Micro/370 loads a new PSW when it takes an interrupt, and this new PSW can prevent other interrupts from interfering with the service routine.

Mode bits control tracing, alternate register sets, instruction and data spaces, and privilege levels. Micro/370 has two sets of registers. Access for reading and writing is controlled by mode bits. Micro/370 also has a Control mode, a System/370 mode, and problem and supervisor states. Each of these is controlled by mode bits. (Some of the bits are not in the PSW in figure 5.29. They are not part of the architecture of System/370.) The MC68000 has bits to control tracing, instruction and data spaces, and supervisor and user states.



Program Status Word High (PSWH)

R	PER mask	(not used by Micro/370)
T	DAT mode	(not used by Micro/370)
IO	I/O mask	Enables I/O interruptions
EX	External mask	Enables external interruptions
Key	PSW key	Storage access key
E	EC mode	Controls PSW format
M	Machine-check mask	Enables machine-check interruptions
W	Wait state	1 = Wait 0 = Running
P	Problem state	1 = Problem 0 = Supervisor
CC	Condition code	Condition code bits
Pgm Mask	Program mask	Enable bits for program exceptions

Program Status Word Low (PSWL)

Instruction address (Program counter)

Micro/370 uses 32 bits
System/370 uses 24 bits

Figure 5.29 System/370 PSW format

The Instruction Registers

In Micro/370, an instruction enters the chip through the data pads and is latched in the IRF. From the IRF, the instruction moves to the IRD, where it is decoded. The IRF is part of the execution unit, and the IRD is at the input to the instruction decoders. There is a 16-bit path from the IRF in the execution unit to the IRD in the state sequencer. The Motorola MC68000 works the same way, except the registers have different names.

Some instructions include operand length fields. The values from these fields must be sent to the execution unit for use in calculating operand addresses. The instruction moves to the IRE from the IRD just before execution begins. At the same time, the length field moves to a special function unit (SFU) in the execution unit. There is a 16-bit bidirectional path between the IRD and the SFU. There is also a 16-bit path from the IRE to the SFU.

Condition Codes

Condition codes are used in the state sequencer for branching. They can be stored in the execution unit or the state sequencer, as they are needed in both places. I put them in the state sequencer. In System/370, the condition codes are bits 18 and 19 of the PSW. The first word of the PSW contains a bunch of bits needed by the state sequencer, so I put the first word of the PSW in the state sequencer.

Communication between Bus Controller and State Sequencer

The bus controller synchronizes and interprets external signals for the state sequencer. Interrupt requests, memory read and write protocols, external error conditions, and reset requests are all external signals. The bus controller informs the state sequencer when a memory read or write completes. For termination of abnormal read and write bus cycles, the bus controller terminates the abnormal bus cycle and reports what happened to the state sequencer, through the Bus Status Register (BSR). There is an agreement between the designer of the state sequencer and the designer of the bus controller that says when the bus controller puts things in the BSR and how long the information stays valid. The BSR is controlled by the bus controller and is observed by the state sequencer.

The Processor Command Register (PCR) is the corresponding register for the state sequencer. This register is controlled by the state sequencer and observed by the bus controller. The PCR holds the bus cycle requests from the state sequencer to the bus controller.

Microcode

The state sequencer sequences the microcode, but microcode directs the actions of the state sequencer and the execution unit. The instructions and the interrupt lines control which microcode runs. The microcode for a microprocessor should use control store space efficiently. The size of the control store is always limited. (Smaller chips are cheaper.) A typical commercial microprocessor will have fifty to one hundred instructions and three to fourteen address modes. Most of the address modes will be valid with any instruction. The product of the number of instructions and address modes is the number of possible control word sequences. If a control word sequence is six or eight states, the number of states in the controller can be large.

Suppose I implement a separate control word sequence for each valid instruction. If the instruction is *w* bits long and all bit patterns are valid, the number of control word sequences is 2^w. If an average control word sequence is *m* microwords, then the control store needs $m*(2^w)$ states. For a 16-bit instruction, the controller will need about a million states. That is the upper bound. I don't need that many. I look at the op code and the address mode and don't worry about the register pointers. I let the register fields select the registers directly.

If there are *k* instructions and *a* address modes, then there could be k*a instruction sequences (if every address mode is valid with every op code). The number of states in the controller is k*a*m. The MC68000, for example, has about 50 instructions and about 14 address modes. Most instructions can be used with any address mode. If instruction sequences average 8 microwords, the controller needs 5,600 states. That is still large.

If the address calculation sequences are shared by the operation sequences, the controller needs only (k+a)*m states. I also might let the op code tell the ALU what to do. ADD, SUB, AND, OR, and XOR might all share the same control word sequence, for example. There is also some sharing of states at the end of

the control word sequences. The MC68000, for example, has only 544 states in the controller. Micro/370 has about 1,024 states in the controller (to do 102 IBM System/370 instructions).

Algorithms

The purpose of a controller is to translate a stimulus (instruction) into control signals for the execution unit. The more capability you put into the execution unit relative to the instruction set, the less complicated the controls are. If you simplify the instruction set enough, you can put everything into the execution unit. Voilà—no controller at all. If you have a multiplier in the execution unit, all the controller has to say is "multiply"—one signal.

The circuit designers would not give me a multiplier. Or a divider. Or a decimal ALU. The Motorola circuit designers would not give them to me for the MC68000, and the IBM circuit designers would not give them to me for Micro/370. Maybe next time I'll get the multiplier or the decimal ALU. But I won't get all I want. The more they give me, the more I add to the list of things I want. As long as you don't have all the hardware you need to do any instruction directly, you need control algorithms.

Neither Micro/370 nor the MC68000 has a multiplier, a divider, or a decimal ALU. Micro/370 does have a 64-bit shifter. The MC68000 has a 32-bit shifter. Multiplication, division, and the decimal operations are done on both microprocessors using simpler ALU operations. Multiplication and division use addition, subtraction, and shifting. The simple decimal Add and Subtract instructions use the binary ALU and a decimal correction factor. The decimal Add and Subtract instructions in Micro/370 are simple special instructions (the nine official IBM System/370 decimal instructions take a whole chip). The algorithms aren't even fancy. Micro/370 and the original MC68000 use a 1-bit Booth's algorithm for signed and unsigned multiplication. For divide, both Micro/370 and the MC68000 convert the operands to positive numbers. Then they divide using an unsigned 1-bit restoring-divide algorithm. Finally, they convert the answer and the remainder to the appropriate sign and check for overflow. Micro/370 also does the System/370 instructions Convert to Binary (CVB) and Convert to Decimal (CVD). These use simple shift and add algorithms.

Pretesting (IBM System/370)

The *IBM System/370 Principles of Operation* says that the CPU may not execute an instruction unless it can get all bytes of the

operand(s) that participate in instruction execution. You cannot store an answer unless you know, before you store the first byte, that you can store all the bytes. You cannot even change the System/370 condition codes unless you know the instruction will complete.

How does that affect the microprocessor? It means you have to figure out ahead of time which operand bytes will be used and make sure you can get to them. This is called pretesting. Pretesting is a requirement for the way virtual memory operation is defined in System/370. (You are not allowed to take page faults in the middle of most instructions.) The OR Character (OC) instruction, for example, reads two operands of length 1 to 256 bytes, ORs them, and stores the answer on top of one of the original operands. I do pretesting for OC by calculating the operand addresses and adding the length field to the result. Then I read the bytes at the end of each operand. If I can get both of them, I begin to execute the instruction from left to right. I have tested the right-most bytes first. Before I write anything in memory or change the System/370 condition codes, I will read the two left-most bytes (as the first part of the operation sequence). This works because page sizes are 2K-bytes or more in System/370.

That did not seem too painful. Consider the Add (A) instruction. That's really easy. The answer goes in a register, and the operand is only one word long. All you have to do is get the operand, add, put the answer in the register, and set the System/370 condition codes. If the operand memory access fails, nothing has been changed yet, so it's easy to handle the access exception.

Now consider the Store Characters under Mask (STCM) instruction. Bytes from a register are placed in contiguous locations in memory under control of a mask. Access exceptions are recognized only for the number of bytes specified by the mask. For pretesting, you have to figure out how many ones are in the mask and test that many bytes. The pretesting is almost as much work as instruction execution.

Trial Execution (IBM System/370)

For a few instructions, you cannot calculate the addresses for pretesting prior to instruction execution. The Translate (TR) instruction is one of those. (So is the Edit and Mark [EDMK] instruction, but Micro/370 does not do that one on-chip.) It is impossible to predict which bytes of Translate's second operand

will be accessed. And you cannot just test the complete address range of the possible second operand bytes because you are allowed to take an access exception only for operand bytes participating in instruction execution. The simplest way to test the operand bytes is to execute the instruction twice, once by trial execution. I practice executing the instruction—without changing any memory locations, registers, or condition codes. If everything goes well, I execute the instruction for real.

ALU Constants

There are not always two operands handed to you when you want to do an ALU operation. Sometimes you must increment the PC by the current instruction length. Sometimes you need a zero. Sometimes you must decrement an index register by one or by two. Where do these little constants come from? In Micro/370 and in the MC68000, the ALU "makes" them.

The ALU and the AU (arithmetic unit) each has a small constant generator attached to one of the inputs. In Micro/370, one of the ALU inputs always comes from the internal B bus, and the other input comes from the internal A bus or from the constant generator. The microcode decides which. The microcode also specifies the value of the constant. See figure 5.30.

	00	01	11	10
00	0 0→alu bd→alu	1 +1→alu bd→alu	3 +3→alu bd→alu	2 +2→alu bd→alu
01	4 +4→alu bd→alu	5 none	7 +7→alu bd→alu	6 +6→alu bd→alu
11	C −4→alu bd→alu	D −3→alu bd→alu	F −1→alu bd→alu	E −2→alu bd→alu
10	8 −8→alu bd→alu	9 −7→alu bd→alu	B ad→alu bd→alu	A −6→alu bd→alu

Transfer	Nanoword Bits
0→alu	0000
+1→alu	0001
+2→alu	0010
+3→alu	0011
+4→alu	0100
none	0101
+6→alu	0110
+7→alu	0111
−8→alu	1000
−7→alu	1001
−6→alu	1010
ad→alu	1011
−4→alu	1100
−3→alu	1101
−2→alu	1110
−1→alu	1111

Figure 5.30 Micro/370 microcode control for ALU constants

Constant	Nanoword Bits	Constant	Nanoword Bits
none	000000		
ag = 0	001010	ag = ba	100001
ag = 01	000001	ag = e8	011010
ag = 02	000010	ag = f0	101001
ag = 03	000011	ag = f2	101111
ag = 06	000110	ag = fa	101101
ag = 07	000111	ag = fe	101100
ag = 08	001000	ag = ff	100000
ag = 09	001001	ag = 100	011011
ag = 0c	001011	ag = 3f00	101011
ag = 0f	100110	ag = ff0f	101010
ag = 10	001100	ag = 80000	100111
ag = 20	001101	ag = 440000	101110
ag = 28	001110	ag = 3700400	011100
ag = 30	001111	ag = 66666666	011110
ag = 40	010001	ag = b1000000	100010
ag = 80	010011	ag = b2200000	100101
ag = 86	010110	ag = b80840ff	011111
ag = 88	010111	ag = ff000000	100011
ag = 8c	011000	ag = fff00000	100100
ag = b8	011001	ag = ffff8000	101000

Figure 5.31 Constants generated by Micro/370

The nanoword decoder to implement the control for the ALU constants is fairly simple. I chose the constants to be in the Karnaugh map so that I could just sign-extend the high-order bit of the field from the nanoword and get a small 32-bit constant. If the nanoword field contains 0101, then I prevent any ALU operation. If the nanoword field is 1011, then I select the internal A bus to the A side of the ALU. For all other cases, the sign-extended nanoword field goes to the A side of the ALU. What about numbers bigger than +7 or −8? More special cases.

Bus General Constants

Sometimes I needed bigger numbers. Sometimes I needed 32-bit masks. System/370 specifies where to store the old PSW and where to read the new PSW for a variety of exception conditions. The addresses are in real memory. Micro/370 has to "make" the addresses and check the new PSW format to be sure it is valid before the PSW is loaded. If it is not valid, it is a specification exception and requires another PSW swap. All this requires several constants and several masks. The masks and constants I used for Micro/370 are shown in figure 5.31.

Any microprocessor needs some constants for starting addresses on power-up and for interrupts. The MC68000 has several constants for interrupt vector addresses. There probably isn't another microprocessor that needs as many constants as Micro/370. Most microprocessor architectures do not have the protection and checking that are part of the architecture of a computer for large systems.

Internal Bus Control

In Micro/370 and the MC68000, two internal buses connect the resources of the execution unit. The buses lie on top of the macros composing the execution unit. Micro/370 has 32-bit internal buses, and the MC68000 has 16-bit internal buses. Operands and addresses travel from sources to destinations on these buses. This sounds easy, but the MC68000 uses 32-bit addresses and keeps a 32-bit program counter. Some instructions provide only a 12-bit displacement or a 16-bit operand. Some instructions manipulate bytes, and some manipulate 32-bit operands. How do I do all that with 16-bit buses? The 12-bit displacement can be sign-extended as it leaves the data input register. The 16-bit values are made 32 bits by dividing one bus into two segments. One segment propagates the 16-bit operand to the low-order half of the destination macro. The second segment copies the sign of the low-order half of the bus and duplicates it for all bits on the high-order bus. It propagates the sign to the high-order half of the macro.

Macros and internal buses in Micro/370 are 32 bits. Most operands are words (32 bits) or bytes. Displacements are 12 bits and must be zero-extended. There are some halfword operands and many byte operands. Micro/370 can zero-extend 8- and 12-bit operands leaving the Data Input register (DI). It can sign-extend halfwords leaving the DI.

When System/370 instructions load a byte into a register, the bytes not being written to must remain unchanged. The Insert Characters under Mask (ICM) instruction loads from 0 to 4 bytes into a general register. Any bytes not being loaded must remain unchanged. I had to be able to store bytes in a general register under control of a mask. I also needed explicit control of which bytes got loaded. Load Address (LA), for example, loads the low-order three bytes and leaves the high-order byte unaffected. Insert Character (IC) puts its operand in the low-order byte of the general register and leaves the other bytes unchanged.

I control the bytes on the bus by having separate bus sense amps for each byte. If the byte sense amp does not turn on, the corresponding byte in the destination macro is unaffected. The buses are true and complement buses. The true and complement rails are both precharged high. If the sense amps do not copy the source information onto the bus, then when the destination macro is connected to the bus, the macro does not lose its value. The microcode controls the sense amps. I can vector the instruction mask field to the sense amp control hardware, or I can specify the byte sense amps in the microcode directly. The default value is for all four byte sense amps to work on each bus.

Bus Interface

The on-chip bus controller is the interface between the (synchronous) internal state sequencer and execution unit and the (analog) external world.

Synchronization

Synchronization is difficult. An interrupt request looks like an analog signal to the bus controller. It looks analog (even though the sending device views the signal as digital) because the bus controller does not know when the signal changes. No big deal. You just sample it (copy it into a register on each occurrence of some clock edge). There is a finite probability that you copied the signal while it was changing. Your latch might not resolve the signal before you check it. Get a faster latch. Maybe there was noise on the line, and you copied a spike. There is no interrupt request, but you think there is. The faster the latch, the more likely you are to copy a spike. Get a slower latch. Now your time to resolve the signal is longer—increasing the probability that you will not resolve it in time. Get a fast latch and compare the sampled value with the current value on the request line. This is better but more expensive and slower. And still you could have an intermediate state.

All that is just for one external signal line. The bus controller looks at a bunch of lines. Micro/370 has 106 signal pads, and all the asynchronous input-request lines must be sampled and resolved. The bus controller decides which requests are valid and synchronizes them for the state sequencer.

External Bus Control

The bus controller runs the bus protocol for talking to external devices (such as the memory and coprocessors). The micro-

processor is the king of the bus. It grants bus cycles to other devices that request bus cycles.

I did not design the bus protocol or the bus controller for either Micro/370 or the MC68000. I'm glad I didn't; it would have been too difficult.

Mode Control

Micro/370 does not do all the instructions defined by the *IBM System/370 Principles of Operation.* (No 370s do.) Because of this, I need a way to re-create the missing parts of the architecture. Micro/370 has a feature called Dual mode. An internal bit called the mode bit divides all operations into two categories: Control and 370. User programs run in 370 mode in 370 space. If the microprocessor encounters an instruction it does not recognize, there is a PSW swap for an operation exception. The PSW swap causes Micro/370 to change to Control mode. Control mode has a separate address space, called control space, with access to 370 space for reading and writing. In Control mode, a program can emulate missing instructions using the available 102-instruction set and adjust locations in 370 space to have the appearance of having executed the missing instruction. The program running in 370 space never knows that the instruction was not executed directly by the microprocessor.

The mode bit cannot be seen or explicitly changed by a program running in 370 space. A program running in control space changes the mode bit with a special load PSW instruction. After the Load PSW, the program resumes in the 370 space.

Shadow Registers

Micro/370 has two sets of general registers because a group wanting to use the registers for emulation of instructions asked us to include them. Both sets are used by the programs in 370 space. One set looks like the general registers and is used for loads and stores. The other set, the shadow registers, keeps a copy of the general registers. Programs running in 370 memory have no control over the shadow registers. They cannot even tell that the extra registers are there. When an exception condition changes the mode to Control mode, the program running in control memory has explicit control over the use of the general registers and the shadow registers. The program can select the following modes:

1. Read from shadow registers; write to general registers
2. Read from general registers; write to general registers
3. Read from general registers; write to general registers and shadow registers (default)

If you are emulating an instruction for a program running in 370 space, you probably will do your work using mode 2. When you are done, you will return the general registers to the values expected by the program running in 370 space using mode 1. The programs running in 370 space will use mode 3 so that there is always a set of saved registers whenever the mode changes to control space.

Special Instructions

Micro/370 has some instructions that are not completely specified by the *IBM System/370 Principles of Operation.* Those instructions all have DIAG (Diagnose) op code '83' hex. The Diagnose instruction is a privileged System/370 instruction that performs model-dependent functions. It can help the program in control space emulate the instructions Micro/370 does not implement. For example, it provides primitive decimal Add and Subtract instructions so the program in control space can emulate the official System/370 decimal instructions more easily. There is a Calculate Effective Address instruction that looks at an instruction's op code and calculates the effective address(es) for the operands. There are instructions that help diagnostics and fake various interrupts; dump the contents of the execution unit; and explicitly control operand read and write spaces (whether operands are accessed in 370 space or control space) and operation of the register sets. None of these instructions operates in 370 mode.

C H A P T E R

6

The IBM Micro/370 Microprocessor

Micro/370 is a single-chip microprocessor. It comprises a clock phase generator, a bus controller, an execution unit, and a processor controller. I specified the execution unit and designed the processor controller. The execution unit represents the programmer's picture of System/370 because it contains the general registers and the PSW. The processor controller translates the official System/370 instruction bit patterns into the appropriate sequence of operation commands for the execution unit and the computer's memory. The bus controller makes the chip work in an electrical environment with memory chips, interrupts, and real-time events. The bus controller synchronizes signals for the processor controller.

Micro/370 is a square chip 10mm on a side. It has peripheral pads (see figure 6.1). To the designers, the chip has a definite orientation: The execution unit is on the bottom, and the bus controller is on the top. Instruction and data addresses go out on the pads around the bottom left corner. The external data bus (EDB) is on the pads around the lower right corner. Bus controls and interrupts occupy the pads going around the top of the chip.

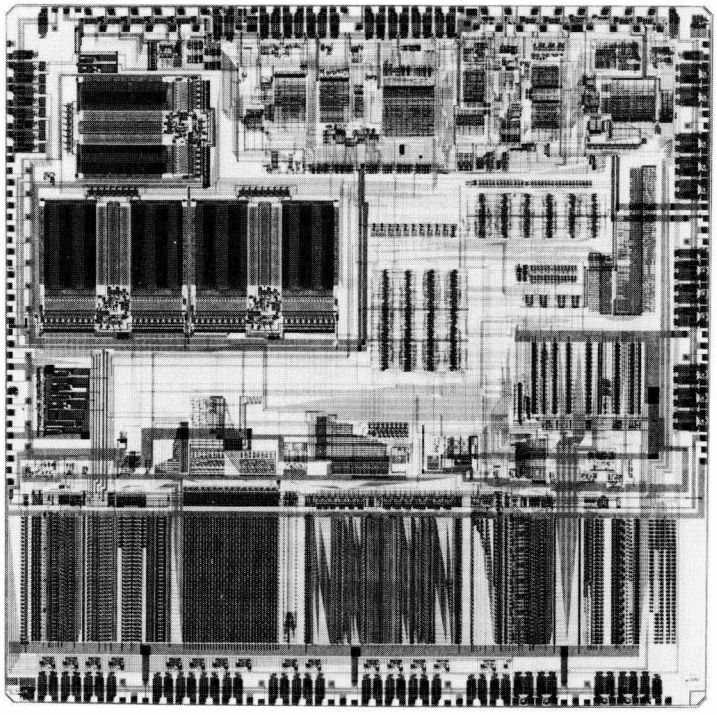


Figure 6.1 Micro/370 chip

Execution Unit

The execution unit is shown in figure 6.2. Notice the bus couplers between AT and AG on the internal A and B buses. These bus couplers are like relays; they divide the buses into separate sections. The bus couplers allow you to transfer data all across the bus or divide the bus into sections for concurrent transfers within sections. The main advantage is the ability to calculate instruction and data addresses using the left section of the bus and simultaneously do arithmetic and data transfers using the right section. But the left section can still use the data pads occasionally, and the right section can use the address pads.

Let's tour the Micro/370 execution unit from left to right:

AO—the 32-bit address output buffer. It consists of the drivers for the address pads. AO gets the addresses from either the A or B bus or directly from AU.

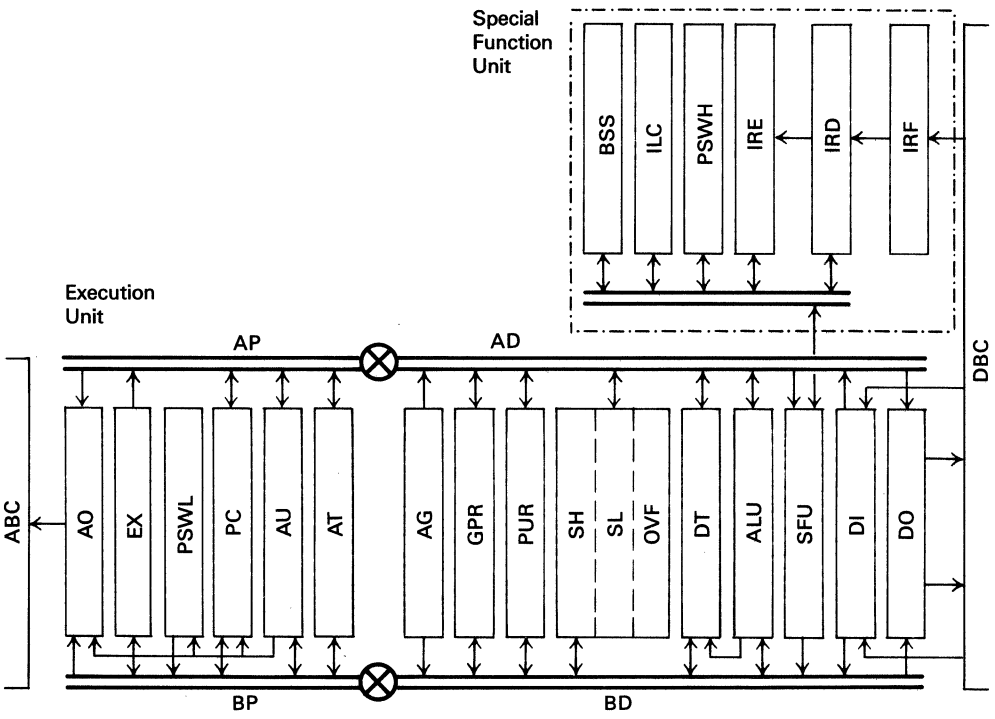


Figure 6.2 Micro/370 execution unit and special function unit

EX—the Execute register. This register remembers the original instruction address during an Execute instruction. It has a read path from the A bus and a bidirectional path to the B bus.

PSWL—the low word of the System/370 PSW. It can send data to the B bus and is automatically loaded from the AU between instructions.

PC—the program counter holds instruction addresses. It has bidirectional connections to the A and B internal buses and a write-only connection from the AU.

AU—the 32-bit arithmetic unit. The arithmetic unit, which only does two's-complement add and subtract, is used primarily for address calculation and instruction address update. Its output is connected (optionally) to either the A or B bus, AO, PSWL, and PC. One input to the AU comes from the A bus. The other input can come from either the B bus or a constant generator. The AU has a result register at its output.

AT—an Address Temporary register connected to the A and B buses.

Bus couplers and sense amplifiers—the sense amplifiers. There is a set for each section of each bus. The individual bytes of the sense amps for the data sections of the bus can be controlled with a mask.

AG—the address generator. This unit sits above the execution unit but has a 32-bit path connecting it to the A or B bus. AG is the source for all the special constants and addresses required by Micro/370.

GPR—the sixteen 32-bit general registers and the sixteen 32-bit shadow registers. The sixteen general registers are R0 through RF. The shadow registers are a copy of the general registers and are called S0 through SF.

PUR—a special-purpose unit used to aid execution of instructions that move nibbles around. (MVN, MVZ, MVO, PACK, and UNPK are examples.) The Pack-Unpack Register (PUR) is connected to the A and B buses, but what happens when an operand arrives at PUR depends on both the bus from which it comes and the instruction that sent the operand. PUR includes an 8-bit register.

SH/SL/OVF—the 64-bit shifter. It does single and double shifts from 0 to 63 bits in one cycle. SH is the high-order half and is connected only to the B bus. SL is the low-order half and is connected only to the A bus.

DT—a Data Temporary register connected to the A and B buses. The ALU can write directly to DT.

ALU—the arithmetic and logic unit. The ALU is used for all general-purpose arithmetic. It can write to the A or B bus or directly to DT. One input to the ALU always comes from the B bus, and the other comes either from a constant generator or the A bus. The ALU has a Result register at its output.

SFU—the special function unit. Here is where I lumped all the odd things I needed to do, including decimal digit and sign checking; the path from the instruction register to the execution unit (needed for getting operand length counts, masks, and immediate values); the path from the bus controller to the execution unit; the path from the state sequencer to the execution unit; the path to the PSW (high-order word); and nibble swaps. The A bus can write to SFU, which can send operands to the B bus. SFU includes a 32-bit register.

DI—the 32-bit Data Input register. Its output can be connected to the A or B bus.

DO—the 32-bit Data Output register. Its input can come from the A or B bus.

BSS—the 8-bit communication register between the state sequencer and the execution unit.

ILC—the instruction length count. ILC holds the length of the instruction being executed. It is located above the execution unit and connected via the internal C bus to the SFU.

PSWH—the high word of the PSW. It is located above the execution unit and connected via the internal C bus to the SFU.

IRE—the instruction register for execution. It holds the first halfword of the instruction being executed and is located above the execution unit and connected via the internal C bus to the SFU.

IRD—the instruction register for decode. It holds the first halfword of the next instruction so decode can begin during execution of the current instruction. It is located above the execution unit and connected via the internal C bus to the SFU.

IRF—the instruction register for fetch. It usually holds the second halfword following the beginning of the current instruction. It is located above the execution unit and connected via the internal C bus to the SFU.

Figure 6.3 is the block diagram of Micro/370. Notice the strong similarity between the block diagram and the actual chip (figure 6.1). This similarity is characteristic of the flowchart method.

Control Store Organization

The Micro/370 control store is divided into two parts. The top part holds microwords that control the state sequencer. I call this the micro control store. The bottom part holds nanowords that control the execution unit. I call this part the nano control store. The control store address is always the same for both parts of the control store. The microword exits the right side of its control store, and the nanoword exits the bottom of its control store (figure 6.4). The control store is just a ROM. The control store address is 10 bits, for 1,024 possible words. Each control store address produces one microword and one nanoword. Microwords are 18 bits. The micro control store is organized physically into two banks of 1,024 10-bit sections. This gives one 20-bit microword each access (18 bits of microword and 2 spares). Nanowords are 71 bits. The nano control store is organized physically into 4 banks of 1,024 18-bit sections. This gives one 72-bit nanoword each access (71 bits of nanoword locations and 1 spare). The control store implements 1,024 nanoword

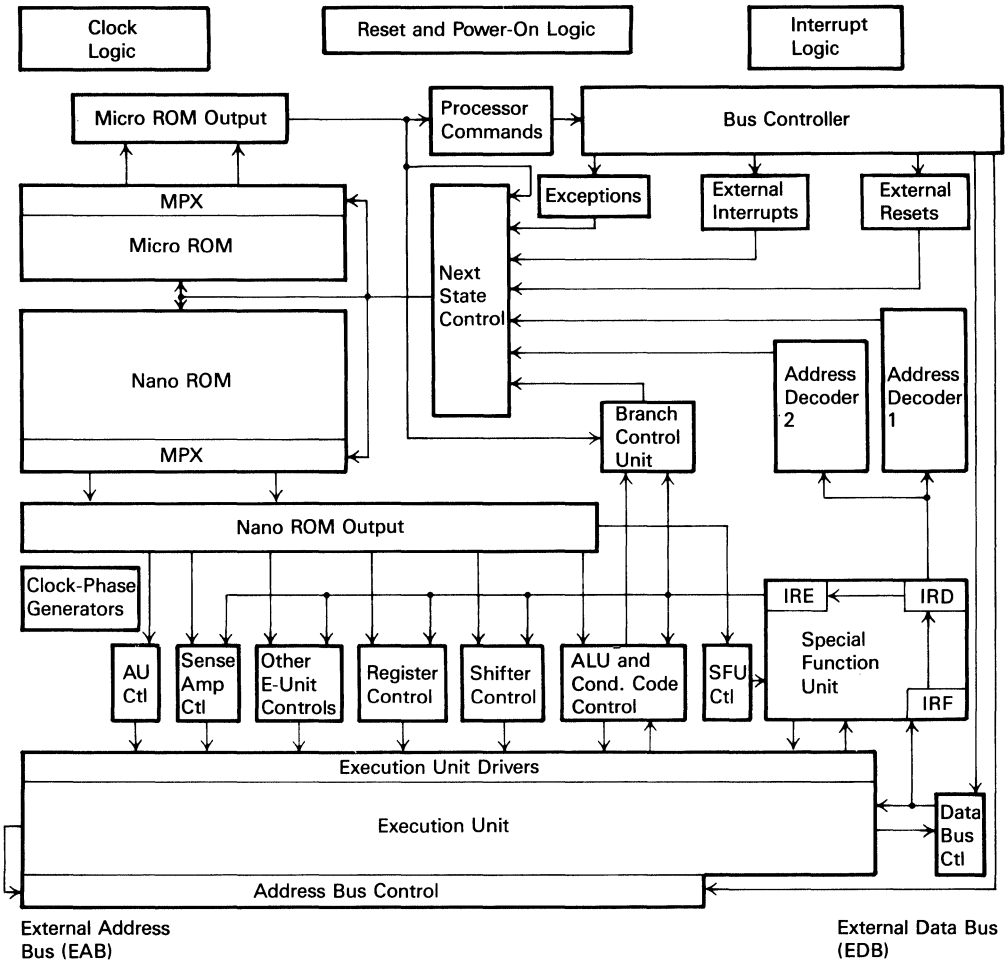


Figure 6.3 Micro/370 block diagram

locations and 1,024 microword locations, but only 984 microword and 984 nanoword locations are occupied (40 control store locations are empty).

The flowcharts for Micro/370 contain 899 microwords and 543 nanowords. There are fewer nanowords than microwords because the flowcharts allow sharing of nanowords. (Nanowords are the tasks in a flowchart box, and the microwords are the rest—sequencing and external bus control.) If the control store

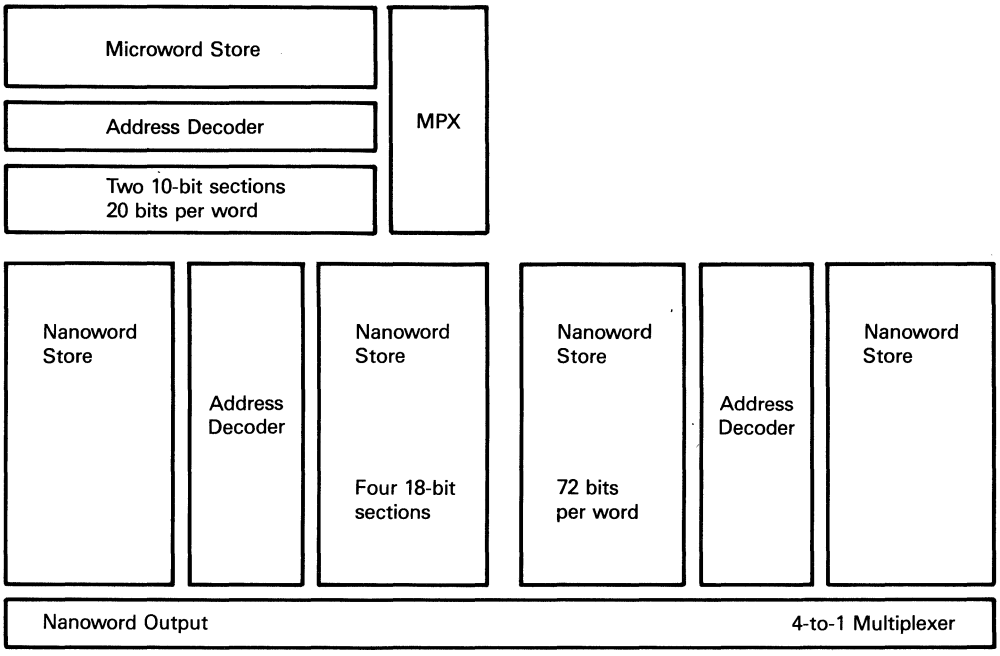


Figure 6.4 Control store logical diagram

implementation takes advantage of nanoword sharing by the microwords, control store size can be reduced by about 30 percent (Stritter, 1978). Motorola implements separate micro and nano control stores and takes advantage of nanoword sharing in the MC680x0 processors. (Motorola designers used the flowchart method for the MC68000, MC68010, MC68020, and MC68030 microprocessors.) The MC68000 control store, for example, contains 544 microwords but only 328 nanowords. Since microwords are 17 bits and nanowords are 68 bits, control store savings is about 32 percent. Micro/370 separates the micro and nano control stores but implements the complete 10-bit address range for each control store. Since the control store address is 10 bits, each control store contains 1,024 words.

Two paragraphs back, I said that there were 984 control words (and 40 empty locations) in each Micro/370 control store. In the next paragraph, I said that there were 899 microwords in the flowcharts. It seems there should be only 899 control words in the control store. There would be only 899 control words if control word addresses were independent. For microcode branches, 2 bits from the branch control unit substitute for the 2 low-order

control store address bits. Microcode branch targets, therefore, must have related addresses. If the same microword is the target of more than one microcode branch, the control word may have to be duplicated in the control store.

Decoders

A1 and A2 are the instruction decoders. They are just above the execution unit and to the right of the control store. The instruction decoders change the official System/370 instruction op codes into addresses for microsequences in the control store. The input to these decoders comes from instruction accesses, which pass through the external data bus connected to the pads around the lower right corner of the chip. The output of these decoders is an address for the control store.

A1, the larger PLA, provides the address for the first microsequence. A2 provides the address for the second microsequence, if any. An instruction with an operand address calculation may have a pointer to the address calculation microsequence in A1 and a pointer to the operation microsequence in A2. A simple register-to-register instruction will have a pointer to the operation sequence in A1 and will be a "don't care" for the A2 PLA.

Next Address Control

The source of the next address to the control store is (usually) determined by the current microword. The microword might provide its own next address. It also might choose an address from A1 or A2. In the event of a microcode branch, the microword will provide a partial address and name a branch condition. The branch unit will provide the last 2 bits of the address. The last microword in an instruction will choose A1—which should be the address of the first microword in the microsequence for the next instruction. If an interrupt is pending, the interrupt will preempt the A1 decoder and provide an address to the interrupt processing microsequence. (This allows interrupts only on instruction boundaries.) A bus error condition or an external reset signal can preempt any other next address selection.

Interface to Bus Controller

The state sequencer is synchronous, and the outside world is not. The bus controller is the interface between the two. The designer of the bus controller and the designer of the state se-

quencer agreed on how they would communicate. In Micro/370, the state sequencer talks to the bus controller through the Processor Command Register (PCR). The state sequencer has to know when to change PCR, and the bus controller has to know when to look at PCR. The bus controller presents status, interrupts, and reset conditions to the state sequencer in the Status Register, the Interrupt Request Register (IRR), and the External Request Register (XRR), respectively. The bus controller knows when it can change these registers, and the state sequencer knows when it is okay to look at the registers. The PCR and Status Register tell when the address and data buses are active.

Figure 6.5 is my view of how Micro/370 looks in a system. To me, everything that is attached to Micro/370 looks like some kind of memory. The state sequencer has to be able to communicate with each of the different types. Sometimes the communication has to be explicit. For example, loading the page tables or the storage keys requires direct communication with the translation buffer or the storage key memory. Sometimes the communication is implicit. ROM, for example, is never addressed explicitly but can be anywhere. (ROM must be there somewhere to provide the starting program on power-up.)

The access indicator field in the microword holds encoded information for the bus controller. It tells the bus controller the following:

- access width (byte, halfword, or word)
- read or write
- service cycle or memory cycle
- instruction or operand
- real address space
- read from write space
- halt
- reset
- no access

When I wrote the flowcharts, I used access indicators as necessary until I had more than sixteen. I thought sixteen would be plenty, but it was not. I had allocated 4 bits in the microword for the access indicator. I changed it to 5. Even that did not seem to be enough. There were too many kinds of reads: reads from instruction space, reads from write space, reads from real memory, and reads from main memory. Each of these could be word or halfword reads, and operand reads could be byte reads.

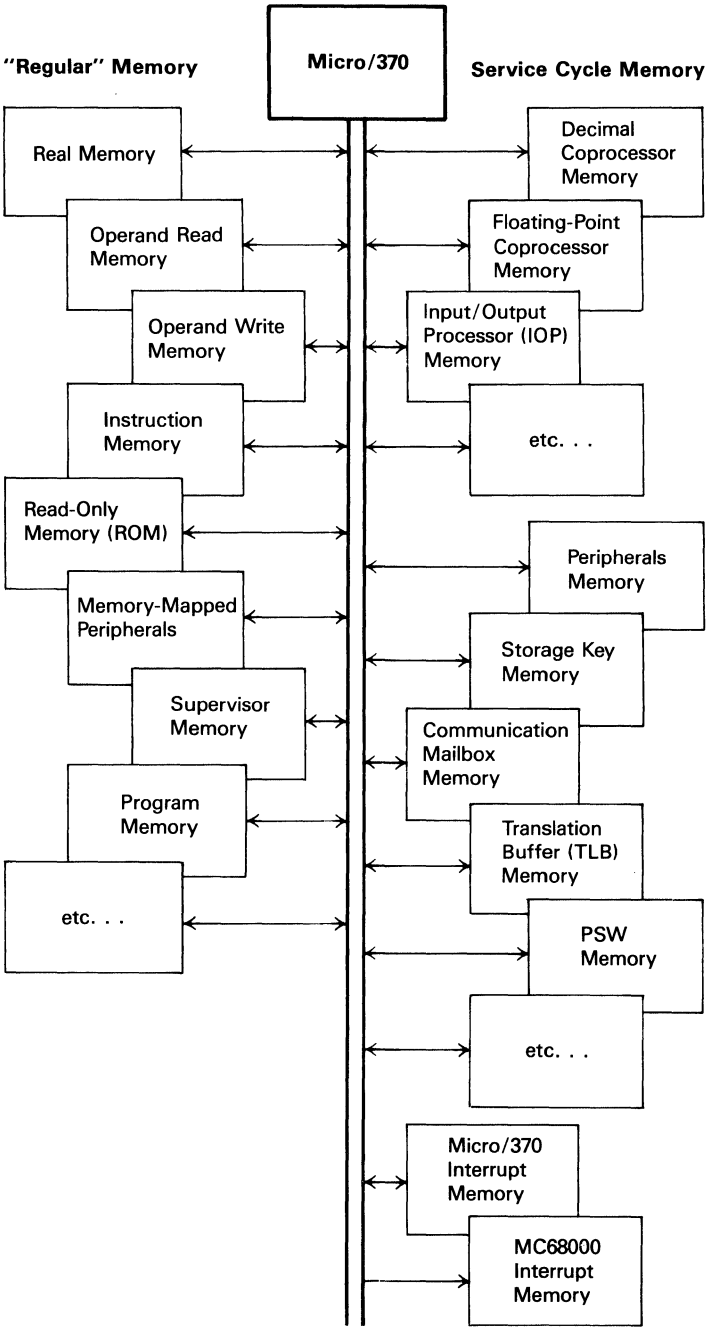


Figure 6.5 Micro/370 "regular" and service cycle memory

These reads could use twenty-four codes. The writes could use another six. Service cycles need at least six, and there are at least three special cycles. I could not get all the access types I wanted to use into thirty-two cases.

Figure 6.6 is the Karnaugh map I used to implement the control lines for the bus controller. It shows how I assigned the values to the access indicators. I was trying to make the decoder simpler. I ended up with thirty different access indicators. Some access types that would be legal cannot be used because there is no space in the Karnaugh map to describe them. There are no split read cycles for real memory or for reading from the write space. The set is not completely symmetrical. At first I used

Karnaugh Map for Function Codes and Access Widths

0

	00	01	11	10
00	0 nacc	1 irin w	3 irin h	2
01	4 frop	5 trin w	7 trin h	6 frin
11	C srmc	D	F srak	E srpg
10	8 swmc	9 wrop w	B swak	A swpg

1

	00	01	11	10
00	0 irop b	1 irop w	3 irop h	2 halt
01	4 trop b	5 trop w	7 trop h	6 rest
11	C trow b	D trro w	F trro h	E trow h
10	8 wrop b	9 wrro w	B wrro h	A wrop h

Instructions

frin		finish read
irin	h w	initiate read
trin	h w	total read

Operands

frop		finish read
irop	b h w	initiate read from operand read space
trop	b h w	total read from operand read space
trow	b h	total read from operand write space
trro	h w	total read from real memory
wrop	b h w	write
wrro	h w	write to real memory

b byte h halfword w word

Service Cycles

srak	read interrupt acknowledge
srmc	read microcode control
srpg	read program control
swak	write interrupt acknowledge
swmc	write microcode control
swpg	write program control

Other

halt	halt state
nacc	no external operation
rest	reset

Figure 6.6 Microword function codes and access widths

whatever I wanted in the flowcharts. Then it came time to make it all fit in the control store. I had to throw out some cases. There were only one or two occurrences of things such as split cycle reads from real memory or from the write space, so I threw them out.

Memory Access Bus Codes

I have a rule of thumb that says I can have two microcycles for every bus access. In Micro/370, I can start a bus cycle during the first microcycle and finish it on the second. States that initiate a bus cycle but do not wait for the result are called initiate cycles. Those that do not initiate the access but must wait for the result are called finish cycles. Those that initiate the bus access and wait for the result are called total cycles.

FRIN—finish read of an instruction. The FRIN cycle does not supply an address to the bus controller. It waits for only one result, since it does not depend on access length.

IRIN—initiate read of an instruction. The IRIN state puts the address on the bus to initiate the instruction read. The state sequencer does not wait for the memory to respond but goes immediately to the next microinstruction, which will have a FRIN function code. IRIN is valid for word and halfword accesses. (There are no byte-length instructions in System/370.)

TRIN—total read of an instruction. The TRIN state sends the address to the bus controller and "hangs" until the instruction is fetched. TRIN is valid for word and halfword accesses.

FROP—finish read of an operand. FROP reads from the current operand read space and is otherwise the same as FRIN.

IROP—initiate read of an operand. The same as IRIN, IROP reads from the current operand read space. IROP is valid for byte, halfword, and word.

TROP—total read of an operand. TROP reads from the current operand read space and is valid for byte, halfword, and word.

TROW—total read of an operand from operand write space. Some instructions read and write the operands (XC, OC, NC, for example). Micro/370 can have separate address spaces for read and write operands. If an operand is read and write, where do you get it (the read space or the write space?) and where do you put it? Brion Shimamoto (my associate in the Micro/370 project) decided that we should get it from the write space and write it to the write space. Even if the operand is write only, you must have a way to read from the write space to be able to test to see whether the location is there before you try to write to it.

TROW permits reading operands from the write space and pretesting write locations. It is valid for byte and halfword.

TRRO—total read of an operand from real memory. TRRO is valid for halfword and word. "Real" memory is assigned special values by the System/370 architecture. In addition, real memory's page is always available, and there are no byte operands in real memory. So I didn't have to do any byte reads from real memory for operands or pretesting. There also are no instructions in real memory (that's why the function code is an operand read). If I didn't need the function code, I didn't assign it.

WROP—total write of an operand. WROP writes to the current operand write space and is valid for byte, halfword, and word.

WRRO—total write to real memory. WRRO is valid for halfwords and words. (There are no byte operands in real memory.)

Instruction accesses use five function codes (five positions in figure 6.6). Function codes appear at the pins. Operand accesses use sixteen. Of those, eleven are for various reads, and five are for writes.

Service (Bus) Cycles

Service cycles (designated Sxxx) are bus cycles used to communicate with all the special devices (e.g., translation buffer, storage key memory, coprocessors, interrupting devices) attached to the Micro/370 bus. Function codes for any service cycle will be 111. For most service cycles, the low-order three address bus bits A[3-1] will be zero. If A[3-1] are zero, the bus cycle is a sense (read) or control (write) service cycle. Sense or control is indicated by the R/W pin. If A[3-1] are not zero, the bus cycle is a Motorola interrupt acknowledge cycle. The MC68000 interrupt acknowledge protocol puts the interrupt level on address bit A[3-1]. How do you know whether you are talking to the storage key memory or to the translation buffer? Micro/370 uses the data strobes—word strobe (WS), upper data strobe (UDS), and lower data strobe (LDS). It is okay to use them because all service cycles are word operations, so they can use the strobes to separate the devices. For this use, the strobe pins WS, UDS, and LDS are called Service Address 3, 2, and 1 (SA3, SA2, SA1) respectively. Any of the data transfer acknowledge signals (BTACK, DTACK, WTACK, or VPA) is a valid response. The flowcharts distinguish the following types of service cycles. These types set the pins to the values listed in table 6.1.

SRAK—service cycle read interrupt acknowledge. SRAK is used to read an interruption code from an interrupting device.

Table 6.1 Sense or Control Service Cycles—Pins

Functions Codes = 111				
R/W	**SA3**	**SA2**	**Sa1**	
x	0	0	0	not used—you cannot tell it is there
x	0	0	1	reserved for programmers using Diagnose
x	0	1	0	reserved for programmers using Diagnose
0	0	1	1	write to coprocessor
1	0	1	1	read from coprocessor
0	1	0	0	not assigned
1	1	0	0	interrupt acknowledge low word
0	1	0	1	new PSW broadcast
1	1	0	1	interrupt acknowledge high word
0	1	1	0	purge translation buffer
1	1	1	0	not assigned
x	1	1	1	storage key memory

SRMC—service cycle read under microcode control. The microcode uses SRMC to read from and signal the special devices attached to the Micro/370 bus.

SRPG—service cycle read under program control. The programmer uses SRPG to read from and signal the special devices attached to the Micro/370 bus. The SRPG service cycle cannot fail. There is no invalid response (even bus error is okay). The response is encoded and passed to the program in general register 1 and the condition codes. The strobes and the address are set by the address calculated in the Diagnose sense service cycle. Program-controlled service cycles allow the programmer to communicate with and control external hardware, such as the translation buffer and the storage key memory.

SWAK—service cycle write. SWAK is used to broadcast the new PSW anytime significant bits change (other than just the condition codes or the instruction address). The high-order word of the PSW is placed on the address pads (its low-order byte is all zeros, so it does not look like a Motorola interrupt acknowledge cycle, despite its acronym). The low-order word of the PSW (the instruction address) is placed on the data pads. (The PSW is always in System/370 extended control [EC] mode.)

SWMC—service cycle write under microcode control. The microcode uses SWMC to write to and signal the special devices attached to the Micro/370 bus.

SWPG—service cycle write under program control. The programmer uses SWPG to write to and signal the special devices

attached to the Micro/370 bus. The SWPG service cycle cannot fail. There is no invalid response (even bus error is okay). The response is encoded and passed to the program in general register 1 and the condition codes. The strobes and the address are set by the address calculated in the Diagnose control service cycle. The programmer could even broadcast a fake PSW, for example.

HALT—request to the bus controller to drive the halt pin (pad).

NACC—no external bus access requested.

REST—request to the bus controller to drive the reset pin (pad).

Execution Overlap

In Micro/370, the execution of the current instruction is overlapped with decoding of the following instruction and the prefetch of the second following instruction. This is true for both microinstructions and instructions. (I call this overlap instead of pipelining because there is still only one instruction in execution at a time.) It is the job of each instruction microsequence to make sure that the next instruction has been fetched sufficiently in advance to allow decoding. The current microsequence also has to fill IRF with the halfword following the next instruction. This allows back-to-back execution of halfword instructions. Inside the chip, while the current microinstruction is being executed, the following microinstruction is being read from the control store and decoded. Fetch, decode, and execute are overlapped at the instruction level. At the microinstruction level, fetching and decoding are overlapped with execution.

Prefetching

It is common for a microprocessor to assume that the next instruction is there, get it, and then throw it away if there is a branch. The MC68000 does this. If you are doing a prefetch while you are decoding a branch instruction, you do not even know you have a branch instruction yet because you are executing the instruction before the branch instruction. How could you know whether the branch will be successful? You can't. You just do the prefetch, and if the instruction (or the memory location or whatever) is not there, the instruction blows up. See figure 6.7.

Your microprocessor is overlapped for fetching, decoding, and execution. While you are executing the Add Register (AR)

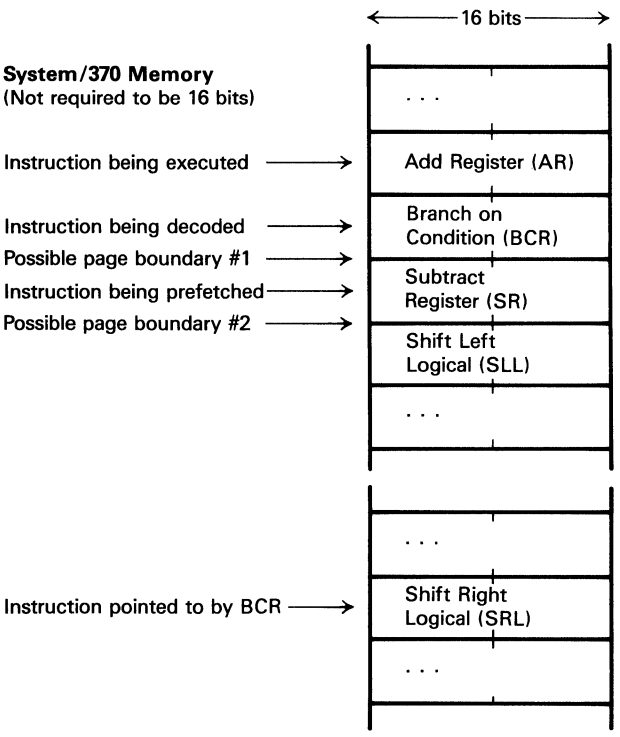


Figure 6.7 How instructions overlap in Micro/370

instruction, you are decoding the Branch on Condition Register (BCR) and fetching the Subtract Register (SR) instruction. That is what the MC68000 does. It is also what Micro/370 does. Further, once the Add Register instruction is done, you begin to execute the BCR. You have three choices for accessing the instruction stream:

1. You can check the branch condition and then access the proper next instruction location.
2. You can access the next sequential instruction since you already have its address, then throw it away if you don't need it.
3. You can assume the branch will be taken, calculate the branch address, and access the instruction at the branch destination. Throw it away if you don't need it.

Notice that the SR instruction is already in the CPU when you begin to execute the branch instruction. It was fetched by the

AR instruction microsequence and is now being decoded. In figure 6.7, I am talking about whether to fetch the shift left logical (SLL) or the shift right logical (SRL) instruction. In the MC68000, I assumed that branches would be taken about 60 percent of the time, so I calculated the destination address and began the prefetch there. Then I threw out whichever instruction I didn't need. (I had both the SR and the SRL.) You have to do another access before you leave BCR because you have to access the halfword following the next instruction. Even if you did not do the extra prefetch, you would have to delay one cycle whenever you branch—to load the next instruction (in this example, SRL) into IRD and decode it. In Micro/370, I used options 2 and 3. Sometimes I got the branch destination instruction, and sometimes I got the next sequential instruction. I got the branch destination instruction for BCR and Branch on Count (BCT), for example. For Branch on Condition (BC), I got the next sequential instruction while I was calculating the branch destination address.

Execution overlap is a good idea, but it is not free. System/370 architecture says: "If you don't need the instruction, it doesn't have to be in memory." On a branch instruction, you are not allowed to take an access exception on the next sequential instruction (because it isn't there) if the branch is taken. "But it's the bus controller that recognizes the access exceptions," you say. Yes, it is. But it's the state sequencer that decides what to do about access exceptions. I designed the state sequencer. In the Motorola MC68000, all the access exceptions were treated alike—the access blew up with a bus error. In Micro/370, all access exceptions are not alike. The bus controller reports the outcome of any bus cycle to the state sequencer in the Status Register. The Status Register is 17 bits—one for each way a bus cycle can terminate. The bits are mutually exclusive. The bits in the Status Register are divided into two groups, normal terminations and exceptional terminations.

For normal terminations, there is no problem. For exceptional terminations, the action of the state sequencer depends on the nature of the exceptional termination. If the exceptional termination results from an operand access (either read or write), then the access is part of the current instruction and you branch to the appropriate exception processing microsequence. If the exceptional termination results from an instruction access, you do not do any exception processing. Instead, you encode the exception condition in a byte and save it.

If the exception occurred when you were loading IRF, save the encoded exception byte in a register called IFX. If the exception occurred when you were loading DI, save the encoded exception byte in a register called DIX. If you were loading both IRF and DI, load the exception byte in both IFX and DIX. If you subsequently do an access that does not cause an exception, erase the byte in the corresponding exception register. When you move the contents of the IRF into IRD to decode the next instruction, you also should move the contents of IFX into IDX. If you try to do an A1 call (using the output of the instruction decoder, which is driven by IRD), or you try to load IRE from IRD and the IDX register is not zero (that is, it contains an exception code), then branch immediately to the exception sequence for the encoded exception condition. If you try to use the contents of DI and the value in DIX is not zero, then branch immediately to the exception sequence for the encoded exception condition.

Figure 6.8 shows how the prefetch protection registers are set for the instruction example of figure 6.7. The three instructions

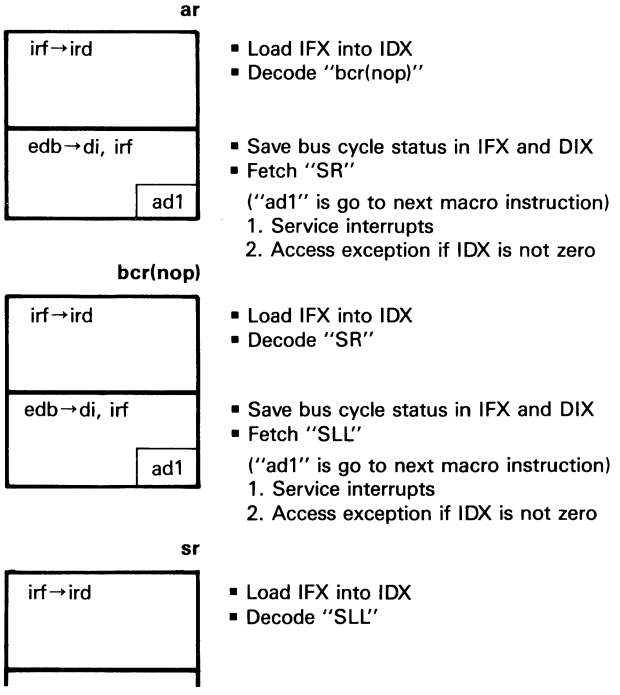


Figure 6.8 Simple instruction overlap in Micro/370

(AR, BCR, and SR) are halfword instructions. If the R2 field of the BCR instruction is a zero, the instruction becomes a no operation (NOP). In figure 6.7, I threw out all the tasks not related to the prefetch protection registers. This sequence of macro instructions illustrates how instruction fetch, decode, and execution are overlapped. Each of the instructions executes in two states. (Rule of thumb: The on-chip state sequencer is twice as fast as the external bus.) During the two states to execute an instruction, a bus access to the halfword following the next instruction is in progress. Here is the process:

A. During the first cycle of AR
   1. Start the external bus cycle to read the SR instruction.
   2. Send BCR (NOP) to the instruction decoder (and decode).
   3. Move IFX to IDX.
   4. Add the operands (begin instruction execution).

B. During the second cycle of AR
   1. Finish the external bus cycle (read the SR instruction into DI and IRF). If this instruction access cannot complete, the bus controller will tell the state sequencer why (via the Status Register). The AR instruction should complete successfully. In this case, even the BCR (NOP) instruction should complete because even that instruction does not need the access that failed. The reason the bus cycle failed is encoded and saved in DIX and IFX.
   2. Store the sum (finish instruction execution).
   3. Decide what to do next.
      a. If there is a pending interruption, read and decode the first microword of the interrupt microsequence.
      b. If IDX is not zero, read and decode the first microword of the access exception microsequence.
      c. Read and decode the first microword of the BCR (NOP) microsequence.

The housekeeping tasks (that's what all the prefetching and decoding are) are exactly the same for BCR (NOP) and SR. If the SR instruction was not available when it was accessed during the AR instruction, IFX and DIX contain the (nonzero) encoding of the status register. When the BCR (NOP) executes, the (nonzero) value of IFX is moved to IDX during the first state. During the second state, the nonzero value of IDX causes the state sequencer to access the first microword of the appropriate access exception microsequence. If there is a pending interruption from

outside the chip, it is given preference. Pending interruptions outrank page faults and other access exceptions. (You wouldn't want to fix the page fault and then take an interrupt and have the interrupt code kick your new page out of memory.) Both AR and BCR (NOP) complete even though an access exception occurred during the AR and before BCR (NOP).

The sequence of figure 6.8 is a good performance indicator for the state sequencer. It shows the processor running back-to-back minimum length instructions. There should be a good balance among bus cycle time, microword access and decode, instruction decode, and execution time. In this example, there is. The bus cycle time is twice the other times. But instruction decode and microword access and decode are sequential, and most instructions take two cycles (compute, then store result). Fetch, decode, and execute are balanced.

Figure 6.9 illustrates how the prefetch protection registers are set and used for instructions longer than one halfword. I have been talking about register-to-register instructions, which are only a halfword long, do not do memory accesses for operands,

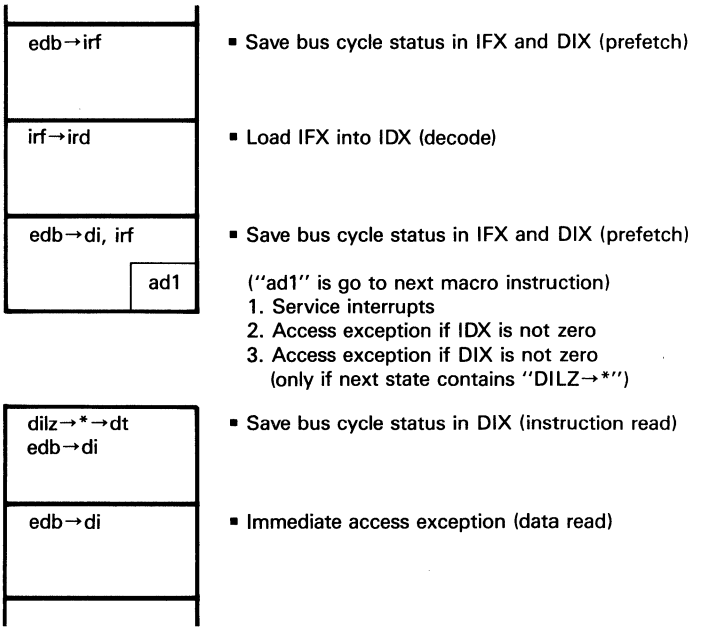

edb→irf
irf→ird
edb→di, irf
ad1
dilz→*→dt
edb→di
edb→di
▪ Save bus cycle status in IFX and DIX (prefetch)
▪ Load IFX into IDX (decode)
▪ Save bus cycle status in IFX and DIX (prefetch)
("ad1" is go to next macro instruction)
1. Service interrupts
2. Access exception if IDX is not zero
3. Access exception if DIX is not zero
(only if next state contains "DILZ→*")
▪ Save bus cycle status in DIX (instruction read)
▪ Immediate access exception (data read)

Figure 6.9 Instruction overlap in Micro/370

and do not calculate operand addresses. If the external bus access fails for any operand read or write, you do not have to save anything (in a prefetch protection register). You cannot complete the current instruction, so you branch to the microcode for the appropriate access exception as soon as you recognize the condition.

Look at the last state in figure 6.9. It is an operand access. If the access fails, you want the next microword executed to be the first microword in the exception microsequence. That's not easy. The external bus access does not complete until the end of the state, but in the interim, the state sequencer is accessing and decoding the next microword in the instruction's microsequence. If the current access fails, you do not want to execute the next microword in the sequence because it might change the state of a register the programmer can see (such as a general register or the PSW). That would violate the requirement of the architecture that an instruction does not affect the programmer's state unless it can complete.

If there is an access exception, the state sequencer issues a signal called "euhold" (execution unit hold) to the execution unit. The euhold signal causes the execution unit to ignore the following microword. (It may do the register reads and internal bus transfers, but it will not change any of the internal registers.) In addition, each time the bus controller returns an access exception status, it ignores the subsequent bus command. (Since the microword accesses are overlapped, the next bus command is issued before the end of the current command and must be ignored if the current command fails.) This one-microcycle "hiccup" by the bus controller allows the state sequencer to issue a new command. If the access exception occurs on a prefetch, the state sequencer saves the encoded status and reissues the original next command. This scheme allows prefetching access exceptions without stopping the current instruction execution. When a prefetch access exception occurs, there is a one-microcycle "hiccup" (one state) in instruction execution time.

The last instruction access in any microsequence loads both IRF and DI (see, for example, figures 6.8 and 6.9). The halfword being loaded is the second halfword following the current instruction. If the next instruction is only one halfword, then the instruction following the next instruction is in IRF. If the next instruction is two or three halfwords long, then the base register

designator and displacement for operand address calculation are in DI. If the current state loads an access exception code in DIX, the state sequencer checks the next microword (the one being decoded) and each subsequent state to see whether the contents of DI will be used. If the execution unit tries to use the contents of DI, the state sequencer will change the next state to the beginning of the exception sequence. This ensures that the exception occurs during the instruction that attempts to use the missing data (as opposed to the instruction that attempted to get the data).

Figure 6.10 is the block diagram of the address decoders controlled by the prefetch protection registers. It shows a simplified

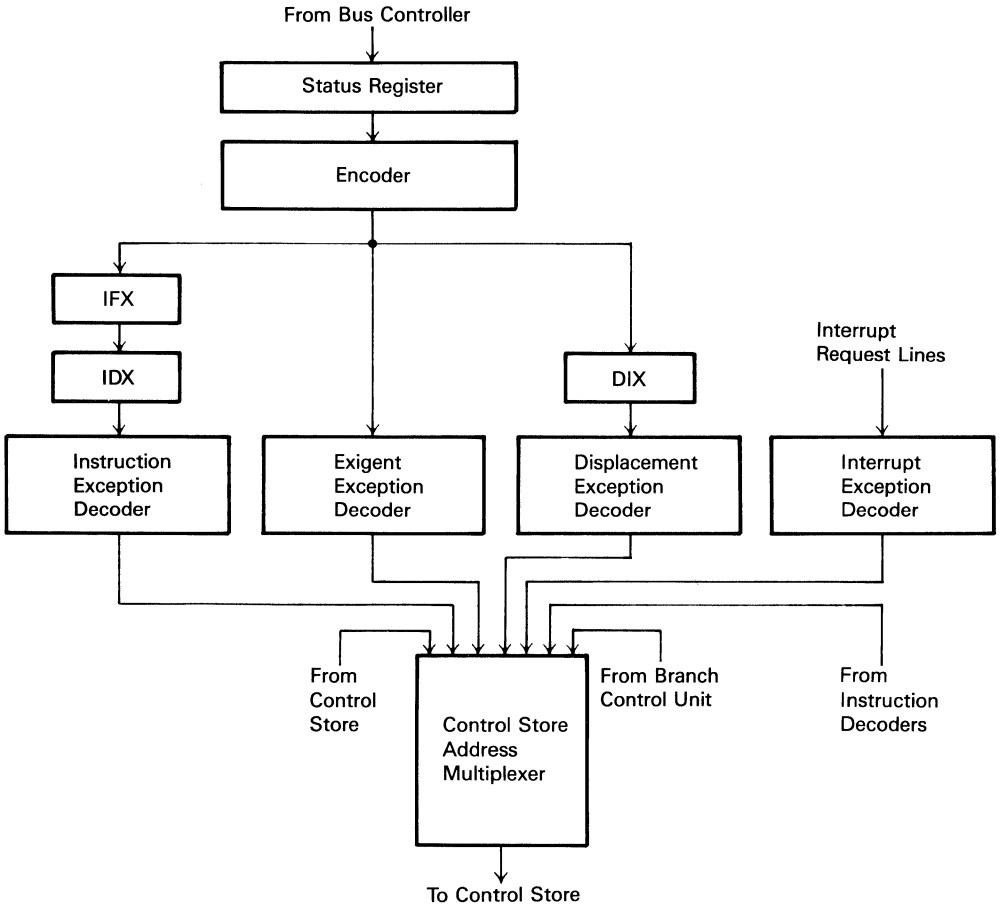


Figure 6.10 Block diagram of prefetch protection registers

multiplexer for the control store next address. Exigent exceptions have the highest priority. Interrupts are second. Instructions are third. Displacement exceptions are fourth. All other control store address sources are mutually exclusive (with each other, not the exceptions) and rank fifth.

Clocking and Timing

Micro/370 uses four-phase clocking. The first phase is the source phase in the execution unit. The operand source gates are opened to the bus. During phase 2, the macros are precharged, and the bus sense amplifiers repeat the source signal on the buses. During phase 3, the destination gate is connected to the bus. The bus signal is strong enough to write over the current contents of the destination register. At phase 4, the results propagate (ALU and condition codes) and the buses are precharged.

Clocking in the state sequencer is limited to the registers and is structured. There is no clocking in any of the PLAs or decoders. The signals are all clocked directly at the registers.

Bus Sense Amp Control

Micro/370 uses sense amps, so the registers can be smaller. The Micro/370 execution unit is bus-oriented. There are two internal buses, and each execution unit macro is connected to these. The buses have both true and complement values. Transfers on the internal buses use a four-phase clock. Suppose a flowchart state contains the task alu→a→r1. At phase 1, a gate is opened to allow the true and complement contents of the ALU output register to connect to the true and complement rails of the internal A bus. At the time the ALU is connected to the internal bus, both rails of the bus should be at logic 1. During phase 1 (while the ALU bits are connected to the bus), the ALU bits cause differential changes in each bus bit. At phase 2, differential sense amplifiers are turned on. (See figure 6.11.)

Suppose you watch the sign bit from the ALU. Assume that it is a logic 1. As it is leaving the ALU, the complement value also is generated. The bus has two rails for each bit and is waiting with both rails at logic 1. During phase 1, each copy of the sign bit (one true and one complement) is connected to its respective bus bit rail. The true rail and the ALU sign bit (for this example)

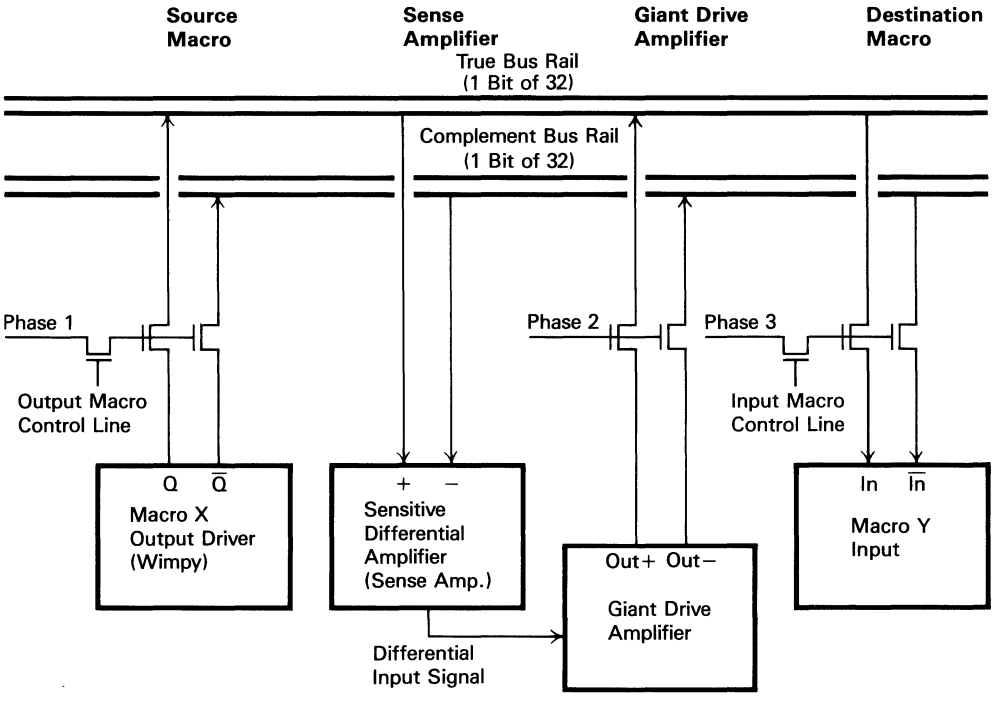


Figure 6.11 Block diagram of 1 bit of bus sense amplifier control

are both at logic 1, so nothing happens. The complement rail is a logic 1 (both of the bus rails always start phase 1 as a logic 1), and the complement of the ALU sign bit is 0. While the gate between the complement bus rail and the ALU output sign bit is open, the complement bus rail value starts to go toward logic 0 (being pulled down by the ALU complement sign bit driver). At the beginning of phase 2, the sense amplifier compares the true and complement rails to see which rail is heading for 0. It then drives the lower rail very fast and very hard toward 0, in this case the ALU complement sign bit. I could not build drivers for each macro capable of driving the entire bus to 0 (or 1) in a short time (because I did not have unlimited space and power for multiple sets of giant drivers). So I built a special differential amplifier that detects what the macro's wimpy driver is trying to do and then uses a giant, fast driver to force the bus to the desired value.

The differential sense amplifiers detect the voltage difference between the true and complement rails for each bit and then

drive the bus in the direction of its tendency. (The tendency is determined by the contents of the ALU bits.) At phase 3, the gate is opened to the register designated by the R1 field of the instruction. The voltages on the A bus simply swamp the R1 register values. This writes the value from the ALU output register into the register designated by R1. At phase 4, the sense amplifiers turn off, and the true and complement rails of the A bus are both driven to logic 1. (This is called precharging.)

The four-phase clocking scheme and sense amplifiers allow the registers to be much smaller because the register output drivers are only large enough to produce a detectable differential voltage on the precharged bus. In effect, all the macros connected to the bus share a common bus driver.

The System/370 registers are all 32 bits. The internal Micro/370 buses also are 32 bits. Some of the System/370 instructions, such as ISK (Insert Storage Key), affect only a single byte of the register. The architecture says the remaining bytes will be unaffected. Other instructions, such as LA (Load Address), affect three bytes of the register and leave the remaining (in this case, high-order) byte untouched. How do you do that with the 32-bit internal buses?

I chose explicit control in the nanoword for the sense amplifier for each bus byte. If you want to transfer only the low-order byte from the ALU output register to the register designated by the R1 field of the instruction, the flowchart task would be alu→a1→r1. The "1" attached to the bus designator tells the sense amplifier control to turn on only the low-order sense amplifier. When the sense amplifier is turned on during phase 2 of the transfer, only the low-order byte of the ALU output register will be repeated on the bus. The other three bytes of the bus will remain at the precharged logic 1 value. When the gate is opened at phase 4, the low-order byte of the R1 register will be overwritten by the value on the A bus. The other three bytes of the R1 register will see only the precharged bus and will not be overwritten. The bus sense amplifiers drive only the low bus-rail, but there is none. The task alu→a7→r1 would cause the three low-order bytes of the ALU output register to be written to the R1 register. If there is no sense amplifier control specified in the task, F (all bytes) is assumed.

System/370 architecture defines two instructions, Insert Characters under Mask (ICM) and Compare Logical Characters under Mask (CLM), which can use more general sense amplifier control. The ICM instruction contains a 4-bit mask. Each bit in the mask corresponds to a byte in a register. The high-order bit corresponds to the high-order byte in the register. The low-order bit corresponds to the low-order byte. The mask can have any value from 0 to 15 (hex F). Contiguous bytes from memory are inserted into the designated register under control of the mask. CLM is similar. It tests contiguous bytes in memory against register bytes corresponding to mask bits. For these instructions, I let the mask field from the IRE control the internal bus sense amplifiers for some states. I form a temporary word with the memory operand bytes in byte positions corresponding to the mask bits. For ICM, I transfer the word to the designated result register with the IRE field controlling the sense amplifiers. The effect is to change only the bytes for which there is a mask value of 1. For CLM, I compare the temporary word with the register word. I mask unused bytes in both words to 0 before I compare them.

Figure 6.12 is a block diagram of the sense amplifier control. In most states, all of the sense amplifiers are running. In some states, the sense amplifiers for the bytes are explicitly controlled by the microcode. In a few states, the microcode gives control of the sense amplifiers to the IRE.

Shifter and Shifter Control

The shifter, like every other element in the Micro/370 execution unit, does only what was demanded by the instruction set implemented. Some System/370 instructions, such as SLDL, shift a register pair, so the Micro/370 shifter is 64 bits. The 64-bit shifter is frequently used as both a 32-bit and a 64-bit shifter. It is used by multiply and divide algorithms for the integer multiply and divide instructions. It is used by Convert to Decimal (CVD), ICM, CLM, and many other instructions.

The microcode uses two basic shift types: explicit shifts and implicit shifts. Explicit shifts tell the shifter exactly what kind of shift and how far to shift. Microword tasks such as "ls1 sh-sl-1" and "rs8 0-sh" tell the shifter explicitly how to shift. "ls1 sh-sl-1" tells the shifter to left shift by one the connected 32-bit sh and the 32-bit sl and to shift a 1 into the vacated low-order posi-

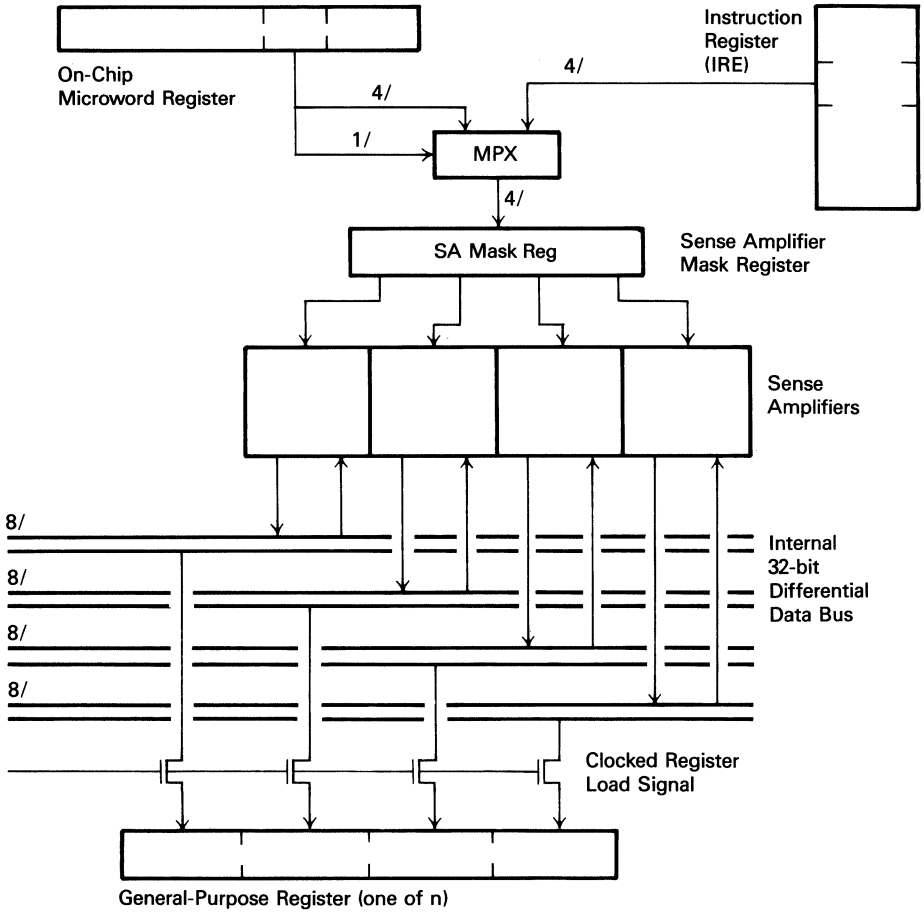


Figure 6.12 Block diagram of bus sense amplifier control by byte

tion. "rs8 0-sh" tells the shifter to right shift 8 bits only the shifter high-order 32 bits and to insert zeros in the vacated positions. Implicit shifts use a microword task of "shift." The shift count is taken from the ALU low-order bit positions, and the direction and shift type are taken from the IRE.

Figure 6.13 is a block diagram of the shifter and the shifter control. Since the execution unit and the internal buses are only 32 bits and the shifter needed 64 bits, it is broken into two (connected) 32-bit pieces. One of the pieces is accessible from each internal bus to allow 64-bit operands to be delivered to the shifter (and shifted) in a single cycle. Figure 6.14 shows the microword

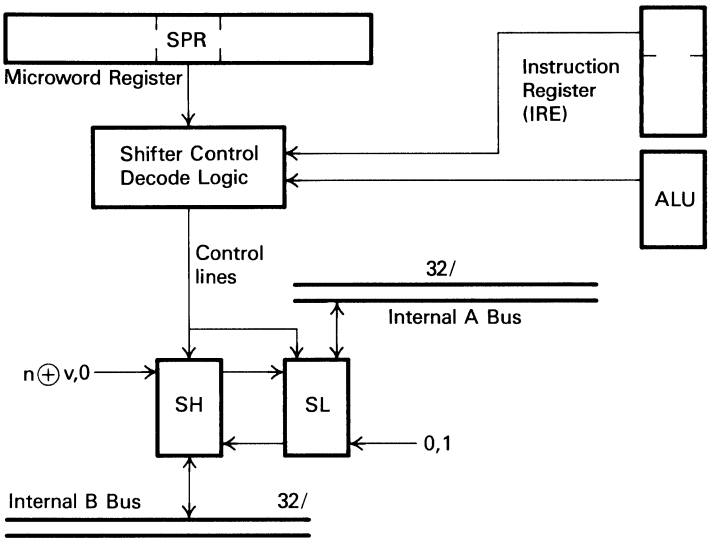


Figure 6.13 Block diagram of shifter and shifter control

SPR 6			
ad→sl	010111	ls8 sh-sl-0;ad→sl	110111
bd→sh	100110	ls16 sh-0	111001
bd→sh1	111000	ad→sl;bd→sh;shift	011111
sh→bd	110001	1xxxxxx→etmssrr	000100
sl→ad	110010	0xxxxxx→etmssrr	001010
sh→bd;sl→ad	110011	x1xxxxx→etmssrr	001100
ad→sl;bd→sh	111111	xxxxx01→etmssrr	000011
rs1 0-sh-sl;ad→sl	100111	xxxxx00→etmssrr	000010
rs1 0-sh-sl;bd→sh	101111	xxxxx11→etmssrr;ad→sl	001111
rs1 n.eor.v-sh-sl;bd→sh	100100	xxx00xx→etmssrr	001001
rs8 0-sh;bd→sh	100101	xxx01xx→etmssrr	001101
rs8 0-sh-sl;bd→sh	101101	xxx10xx→etmssrr	000001
rs16 0-sh-sl	101001	xxx11xx→etmssrr	000101
rs16 0-sh-sl;bd→sh	101100	0011101→etmssrr;ad→sl	000111
rs32 0-sh-sl	101000	0x00011→etmssrr	001011
ls1 sh-0	111011	ad→pur	010000
ls1 sh-sl-0;bd→sh	111100	bd→pur	011101
ls1 sl-sh-0;bd→sh;ad→pur	011110	pur→bd	010101
ls1 sh-sl-1;bd→sh	110101	pur→ad	011011
ls1 sl-sh-0;bd→sh	111110	ad→pur;pur→bd	010100
ls2 sl-sh-0;bd→sh	110110	bd→pur;pur→ad	011001
ls4 sh-sl-0;bd→sh	111101	bd→pur;ad→pur	011100
ls8 sh-0	111010	purh→ad;sh→bd	010011
ls8 sh-0;bd→sh	110100	purl→ad;sh→bd	010001
ls8 sh-0;bd→sh1	110000	none	000000

Figure 6.14 Flowchart tasks and microword control assignments

tasks and bit assignments for shifter control. The shifter shares a 6-bit microword control field with the mode control bits and the Pack-Unpack unit (PUR). Mostly, these units are mutually exclusive in use. If they are not, joint uses become cases, too. For example, the task "ls1 sl-sh-0;bd→sh" differs from the task "ls1 sl-sh-0;bd→sh;ad→pur" even though the shifter operation is the same for both.

References

Skip Stritter and Nick Tredennick, "Microprogrammed Implementation on a Single Chip Microprocessor," in *Proceedings of the 11th Annual Microprogramming Workshop*, November 1978, pp. 8–16.

C H A P T E R

7

Hardware Flowcharts for Micro/370

In this chapter, I describe the flowcharts for Micro/370 in detail. I drew the first execution unit on 7 January 1981. I read the *IBM System/370 Principles of Operation* and guessed at the execution unit I would need to implement the microprocessor. The execution unit had two internal buses to which all macros were connected. The macros are 16 bits. Why did I choose 16-bit macros? I did not have a circuit designer to consult (there were only two of us at the time), and we had not selected the technology. I assumed the technology would be similar to that used for the Motorola MC68000— nMOS with 3 to 3.5 micron channels and a single level of metal. I knew that two buses could lie (in metal) on top of the layout of the execution unit macros but that three buses would cost more area. If the buses were to be metal, the control lines (which run perpendicular to the buses) had to be in poly. The sheet resistivity of the poly was low enough to permit 16-bit macros, but it was too high to use 32-bit macros. (The propagation time was too long, limiting the speed of the execution unit.)

My reasoning didn't work out. When Mon Yen Tsai, a circuit designer, started working on the project, he chose an IBM technology with two levels of metal. I could run the control lines on metal, allowing 32-bit macros with no speed penalty.

The execution unit went through about twenty-three revisions before the end of the project. That's typical—and characteristic—of the flowchart method. You do the flowcharts to make the state sequencer execute the instructions. If you need to add hardware or move something, you just do. All the pieces of the design are treated the same way. You worry first about how to do something, then about how to implement it. You do not fix the controller or the execution unit and try to implement an instruction set on the fixed hardware. The implementation will not be as good as it might be. When you fix structures in advance, you are guessing what resources you will need. It is not likely that you will get it exactly right. (Engineers design with fixed structures because that is how most textbooks portray the design methods. The texts are wrong.)

Start with the execution unit, the flowcharts come next, and then you add detail. When the flowcharts are done, use them to design the hardware.

Practice Level 2 Flowcharts

The first flowcharts I have for Micro/370 are dated 29 January 1981. They are preliminary level 2 flowcharts. I did them on scrap paper using the execution unit dated 8 January 1981. I skipped the level 1 flowcharts and began with level 2 flowcharts because I wanted to see whether my initial execution unit was reasonable. (I would do level 1 flowcharts once I had a reasonable execution unit.) The first instruction I did was BCR (Branch on Condition Register) with the R2 instruction field zero. Why did I choose that one? In IBM System/370, a BCR instruction with a zero R1 or R2 field is a "no-operation." I wanted to do the simplest instruction to see whether I could overlap instruction fetch, instruction decode, and instruction execution. (If I could not, I would change the execution unit immediately.) It worked. I could make a series of no-operations work back to back—one being fetched, one being decoded, one being executed, all at the same time. I did BCR and BC (Branch on Condition) on 29 January.

On 30 January, I did microsequences for the System/370 address modes and the LA (Load Address), L (Load), LR (Load Register), LTR (Load and Test Register), LH (Load Halfword), and ST (Store) instructions. Everything was okay until I got to ST. The execution unit I had would not efficiently calculate the address for the RX (memory-to-register) address mode form where

the X2 (index register) field was not zero. On 30 January, I worked on the execution unit to fix the problem. I moved things around and added more bus couplers. I redrew the execution unit on 2 February. Then on 2 and 3 February, I again did practice level 2 flowcharts for those instructions plus the TM (Test under Mask) instruction. For TM, I changed the execution unit. I added SFU (the special function unit) to get the mask value from one of the instruction registers into the execution unit. On 4 February, I did MVC (Move Character). I changed the execution unit again. This time I added the AT (Address Temporary) register to save one operand address while I was incrementing the other.

MVC was the tenth instruction. Is there a significance to this set of instructions? Yes. It was my associate Brion Shimamoto's guess at the top ten (frequency of occurrence) instructions in a typical Micro/370 program. They dominate performance, so the execution unit must do them well.

I drew the original execution unit on scrap paper and transferred it (adding detail) to an 11-by-17-inch sheet of vellum. I would have preferred 17-by-22-inch light blue vellum with a blue grid at ten lines per inch, but I had to take the smaller paper that was available in the stockroom. It turned out to be too coarse, so I later changed to an 11-by-17-inch green vellum with a green grid also available in the stockroom. The pencil I prefer is a Pentel .5mm mechanical with F lead. Softer lead (or coarse paper) causes the lead to smear. Harder lead will break too easily when you press hard enough to make dark lines. I use three erasers: an electric eraser with a metal erasing shield for close work, a WR Jet (about a 4-inch eraser in a plastic pencil-like holder), and a Pink Pearl for ordinary erasing. You should use the vellum with the grid to the back so you do not erase it as you make changes.

I do virtually all my initial work on used paper and then transfer it to vellum. Once I have the original vellum drawing, I make reduced photocopies of it to write on. I make (and so label) an edit copy of each drawing, which is my working document. Once I decide on a set of changes, I update the vellum copy and produce a new reduced edit copy. I recycle used paper for flowcharting, too. For flowcharts, I load the copier with used paper and run off a grid on the blank side. The grid for practice flowcharts is four across (each 2 inches) and six down (each 1½ inches). I have used this grid size for several years; it works well. Once Dick Hadsell (our design automation engineer) got

the flowchart drawing program working, I got flowcharts with seven boxes to a page vertically in six columns. I experimented with a new grid size, thinking it would conveniently match the computer-generated grid. It had about twice as many boxes as my usual grid, but the boxes were too small for writing tasks with ordinary size print.

Level 1 Flowcharts

I began working on a format for level 1 flowcharts on 13 February 1981. I began working on level 1 flowcharts the next work day. It took me about two and a half months to do level 1 flowcharts for about seventy System/370 instructions. I did two revisions of the execution unit, but the changes were relatively minor. I added the DT register and made the AU do subtraction (for the MVCL instruction) as well as addition. I changed the special path from AU to AT to be a special path from AU to PC. I added a special path from ALU to DT. I combined DT with the ALU for double-length shifting.

Official Level 2 Flowcharts

I began official level 2 flowcharts on 29 April 1981. In the next two days, I did eighteen System/370 instructions. On 30 April 1981, I drew the first page of level 2 flowcharts.

Once I had a page of level 2 flowcharts, I started a card file of states. I got a package of lined 3-by-5-inch cards from the stockroom and made a card for each level 2 flowchart state. I sorted the states according to external bus access type (FRIN, FROP, IRIN, IROP, NACC, TRIN, TROP, TWOP). I arranged each set alphabetically by task. (The tasks within a state are already alphabetized.) The cards helped me find similar states so that I could eliminate states. (I sorted by external bus access type because I could not combine states with different external bus activity.) I eliminated five states (of the forty-five on the page) by making the cards. Over the next month, I did level 2 flowcharts for about sixty-six System/370 instructions. Then came a big change in the execution unit.

On 2 June 1981, I met with the circuit designers for about three hours to discuss the technology for Micro/370. The circuit designers chose an IBM technology that permitted two levels of metal. My original assumption that the control lines for the exe-

cution unit would be in poly was wrong. Based on this assumption, I had limited execution unit macros to 16 bits. Otherwise, the resistive delay on the control lines would limit execution unit speed. With two levels of metal, one level could be used for the control lines. The macros could be 32 bits tall without slowing the execution unit. On 2 June 1981, I redrew the execution unit with 32-bit macros. Changes to the flowcharts to accommodate the new 32-bit execution unit were minor (less than a day's work).

The ALU and condition code (CC) control table first appears with revision 5 of the execution unit. I added it because I was trying to figure out how to set the condition codes for the LCR (Load Complement Register) instruction and found out I needed two different ALU operations. The op code by itself can pick only one. I already knew from experience with the Motorola MC68000 that I would need such a table. We could not build the hardware to do instructions such as Multiply and Divide directly. They have to be done with algorithms that include more than one ALU operation, such as shifting, adding, and subtracting. If the instruction op code can pick only one ALU operation, microcode has to select the others.

The table works like this: The instruction op code picks the row, and the microcode picks the column in the table. This means that the instruction op code does not really pick an ALU operation; it picks a set (the ones you fill in its row). You can let different instructions pick different rows. Then, if the way they store and access operands is the same, they can share the same microsequence. In the original Micro/370 flowcharts, fourteen System/370 instructions share the same microsequence (A, AL, ALR, AR, N, NR, O, OR, S, SL, SLR, SR, X, and XR).

When I added the ALU and CC control table, I didn't worry about how to implement it. I was writing down what I wanted to be able to do. When the circuit designers asked what I wanted for Christmas, I showed them the table. I knew they might not be able to give me exactly what I wanted, but if I didn't ask for exactly what I wanted, they wouldn't know what to shoot for. (Before I can seriously show the table to circuit designers, I have to translate it into some kind of logical representation. I didn't worry about that yet, though). This process allows me to document the requirements of the functional unit that best suits my needs. I assume that it can be implemented. That is inventing: Work with details to understand exactly what you need; make

up a way to do it; write down what you want; then figure out a way to build it.

I did flowcharts for the shift instructions (for the third time—once as 16-bit level 1 flowcharts, once as 16-bit level 2 flowcharts, and finally as 32-bit level 2 flowcharts) on 19 June 1981. That's when I added a 64-bit single-cycle shifter (with overflow detection) to the seventh revision of the execution unit. With execution unit revision 7, I began distributing reduced copies of the flowcharts, which I attached to a copy of the execution unit, to all the people on the project.

Between published revisions, I wrote flowcharts for new instructions or rewrote flowcharts for instructions I had done. The number of revisions I made before I published a new set of flowcharts varied, depending on how big the changes were and how cluttered my edit copy was getting.

In revision 8 of the execution unit, dated 4 August 1981, I included an address generator (AG), which I used to supply the constants and addresses for assigned storage locations used by Micro/370. I added an ILC (Instruction Length Count) register and two 32-bit registers to hold the 64-bit System/370 PSW. I added PSWL (PSW low) close to the PC and PSWH (PSW high) above the execution unit because it collects state information from various sections of the microprocessor.

I added PUR (the Pack-Unpack Register) to the execution unit with revision 9, dated 3 September 1981. This was the last major change in the execution unit.

Sample Instructions

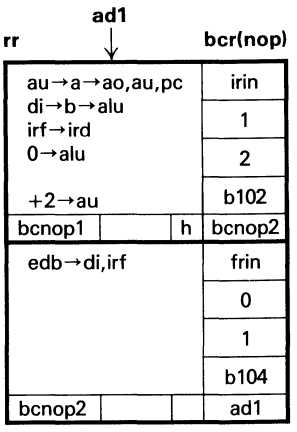
Here are some System/370 instructions to illustrate the Micro/370 flowcharts. The sample flowcharts are taken directly from the Micro/370 data base using the flowchart drawing program. I used a text editor on the IBM PC to line up all the states in a single column. I have added comments to explain what is happening in each state (something I don't do in the real flowcharts).

Branch on Condition Register

Branch on Condition Register (BCR) with the R2 field zero is the simplest of the System/370 instructions because it is a NOP. Sometimes tasks with no apparent function appear in a state. See di→b→alu and 0→alu in state bcnop1 in figure 7.1, for ex-

BCR Instruction Format

BCR	M1	"0"



[ad1] the A1 decoder points to this state
[rr] instruction format is RR/[bcr (nop)] IBM System/370 instruction(s)
[irin] initiate read of an instruction

- Prefetch next instruction halfword, save PC
- Not used in this instruction (see below)
- Load instruction decoder with next halfword
- Not used in this instruction/[2] two other microwords use the nanoword/[b102] page b column 1 location 2
- Increment instruction address/[bcnop1]the state name/
  [h] halfword access request to bus controller/[bcnop2] the next state
- Put prefetched instruction in DI and IRF

[frin] finish read of instruction
[0] ALU not used in this instruction

[ad1] get next microaddress from A1 decoder (A1 is an instruction decoder)

Figure 7.1 Instruction format and flowcharts for BCR

ample. This happens because states are shared among microsequences. Tasks not useful for BCR are necessary for L (Load), the microsequence that shares state bcnop1 with BCR.

Figure 7.1 is the flowchart for the BCR instruction with the R2 field zero. (In System/370, this is a NOP.) Figure 7.2 is an illustration of the relationship between the memory representation of the program we are running (in this case, all BCRs with the R2 field zero) and the contents of execution unit registers. Just before beginning to execute BCRa, PC points to BCRb and AU points to BCRc. IRE contains BCRa and IRF contains BCRb. To overlap fetch, decode, and execute, we have to decode BCRb and fetch BCRc while BCRa is executing.

Load Multiple

Figure 7.3 illustrates Load Multiple (LM). This instruction loads the System/370 general registers, beginning at the register designated by the R1 field through the register designated by the R3 field, from memory starting at (B2)+D2. It shows microcode looping and conditional branching. It also shows pretesting. You cannot change registers, the PSW, or memory (no official state bits) unless you can run the instruction to completion. Each instruction, therefore, checks to make sure all the operand bytes it needs will be available before any state bits are changed. For

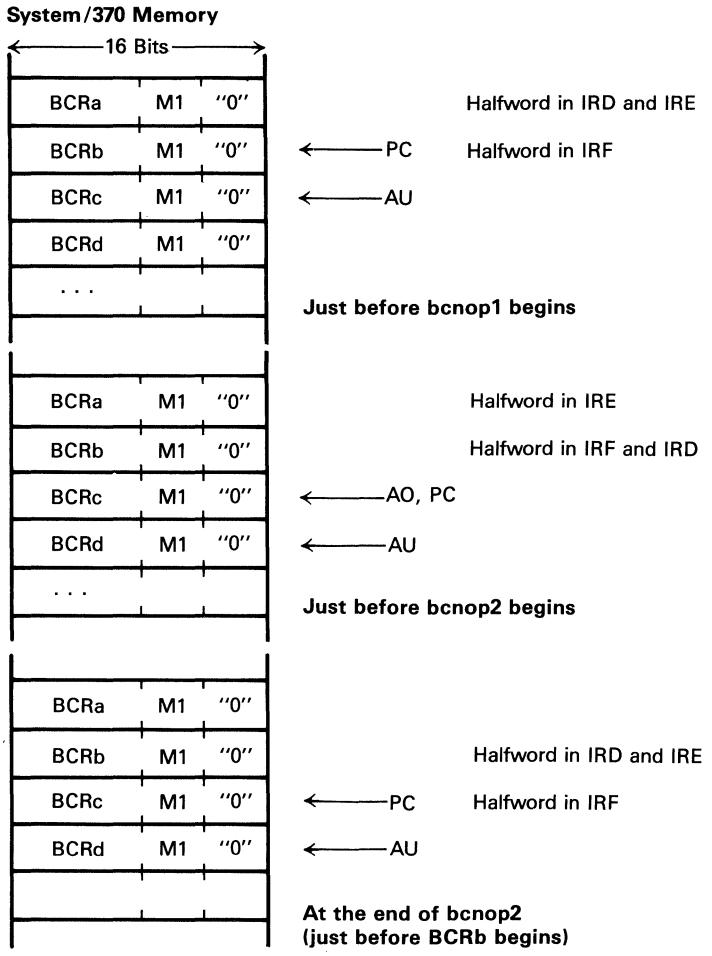


Figure 7.2 Inside the execution unit (executing BCRs)

load multiple, this means you have to figure out how many registers the instruction will read and then read the first and last bytes before you change any registers. This example shows how an address calculation sequence is shared. The same address calculation sequence also is used by the Store Multiple (STM) instruction.

LM goes from the register designated by R1 through the register designated by R3. If the R1 register is larger than the R3, it wraps around. You can load general register 13 through general register 7, for example. If the R1 register is the same register as the R3 register, the instruction loads one register. Subtracting

the R1 register number from the R3 register number gives a count one less than the number of registers to be loaded. Add three to four times the difference to get the byte offset to add to (B2)+D2 for the address of the last byte accessed. This works if the R3 register number is bigger than the R1 register number. What if it is not? The subtraction will give a negative result. If it is ANDed with 'ØF' (hex) (made a positive modulo 16 result), it gives the correct result—a number one less than the number of registers accessed. The starting address can be any byte boundary in memory (see figure 7.3). Figure 7.4a, b shows flowchart sequences for addressing and execution.

Translate

I chose the Translate (TR) instruction because it shows trial execution. You cannot change any System/370 state unless you know that the instruction can complete. You have to pretest the operand bytes to be sure you can get them all. You cannot just check the maximum range of operand fetches because bytes that do not participate in the instruction are not required to be there. The way I check to see whether I can execute the instruction is to practice. I get all the bytes as if I were executing the instruction, but I don't change any state. If I can get all the bytes, I execute the instruction.

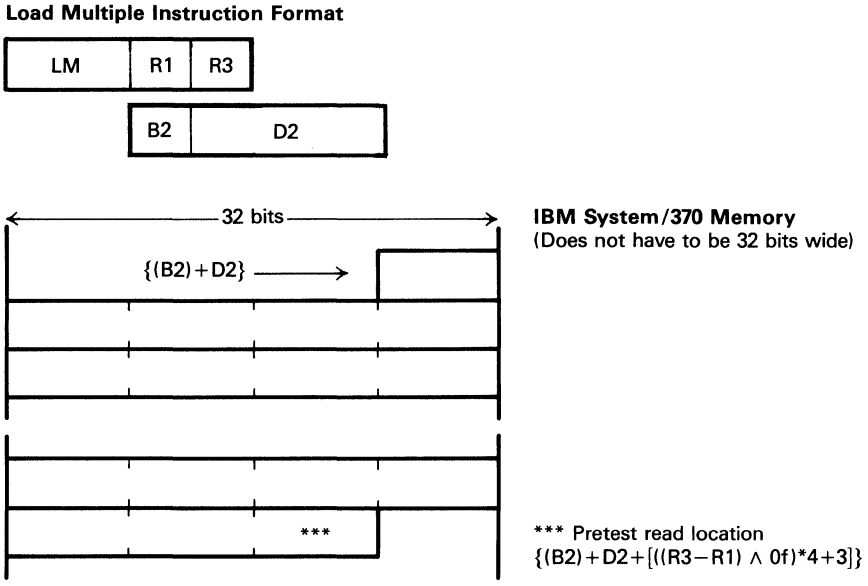
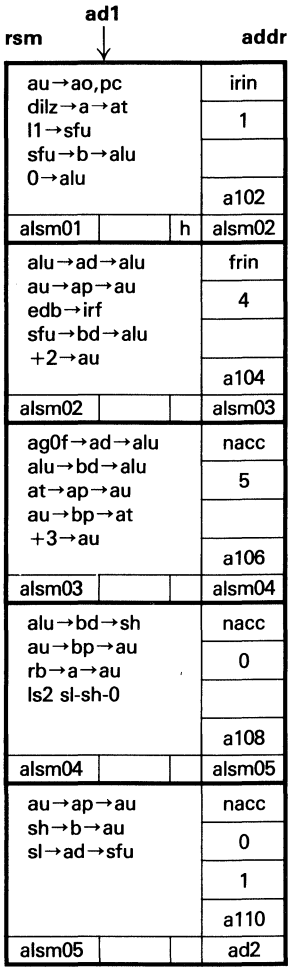


Figure 7.3 Load Multiple (LM) instruction format and memory operand format



[ad1] call loads the R3 field into SFU
[rsm] RS format for load and store multiple
[addr] address calculation sequence

- Save PC, initiate prefetch
- Zero extend displacement D2; save in AT
- Get the R1 field from IRE[8-11]
- Send the R3 field to the ALU

- R3 to ALU on a bus (data section only)
- Increment program counter

[4] ALU function A-B; condition code not set

- R1 field to ALU on b bus

[nacc] no access (no external bus request)

- 0000000F to ALU (makes ALU positive, mod 16)
- (R3-R1) to ALU

[5] ALU function AND; condition code not set

- Save updated program counter
- Add 3 to the (zero extended) displacement; ALU contains one less than the number of registers to be loaded or stored

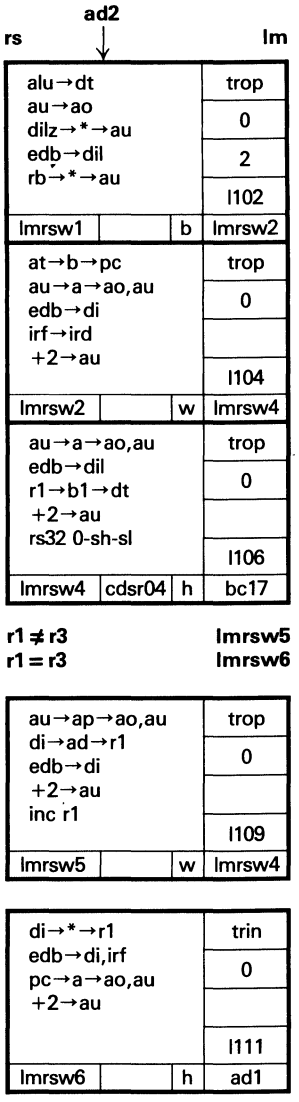
- Send register count to shifter
- Add D2+D3 to B2 (address of low order byte in first location)
- Multiply register count (less 1) by 4 (to get 4 less than bytes to be accessed)

- Add byte count (less 4) to D2+B2+3 to get address of last byte to be accessed
- Not used in this instruction

[ad2] get next microaddress from A2 decoder

Figure 7.4a Address calculation flowchart sequence for LM

Here's what Translate does. (Refer to figure 7.5.) Read the byte at (B1)+D1. Zero-extend that byte and add it to (B2)+D2. Read that byte and store it in (B1)+D1. Read the bytes at (B1)+D1+i in order from i=0 through i=l and translate them in this manner. Addresses in the second operand could be anywhere in the range (B2)+D2+0 to (B2)+D2+255, but I don't know how many there are or where they are. There could be 256 different addresses, or they could be all the same—and that one address could be anywhere in the range.



[lm] Load Multiple instruction
[trop] total read of an operand
- Not used in this instruction
- Read last byte to be accessed (pretest)
- Zero extend displacement D2
- Load DI (low order halfword) with byte read
- Add B2 to zero extend displacement D2

[2] two other microwords use the nanoword
[b] byte access request to bus controller

- Save updated program counter
- Read first halfword to load
- Load into both DI halfwords
- Move prefetched instruction to IRD
- Increment operand address

[w] word access request to bus controller

- Read second halfword to load
- Load halfword into DI (low order halfword)
- Not used in this instruction
- Increment operand address
- Not used in this instruction

[cdsr04] nanoword used by this microword
[bc17] branch condition 17
Continue reading
Last word has been read ([R1=R3] checks IRE[8-11]=IRE[12-15])

- Read next operand halfword
- Put previously accessed word in R1
- Load next halfword
- Increment operand address
- Increment R1 field IRE[8-11] for next store (cannot affect current register store)

[trin] total read of an instruction
- Store the final register to be loaded
- Load prefetch instruction in di and irf
- Prefetch second following halfword
- Increment program counter

[ad1] get next microaddress from A1 decoder (decoded from IRD loaded in lmrsw2)

Figure 7.4b Flowchart sequence for LM execution

To be sure all the bytes of the first operand are there before you change anything, you could read the bytes at (B1)+D1 and (B1)+D1+l. But how do you check the bytes of the second operand? You have a maximum range of 256 bytes (the biggest offset you could get from a byte read from the first operand

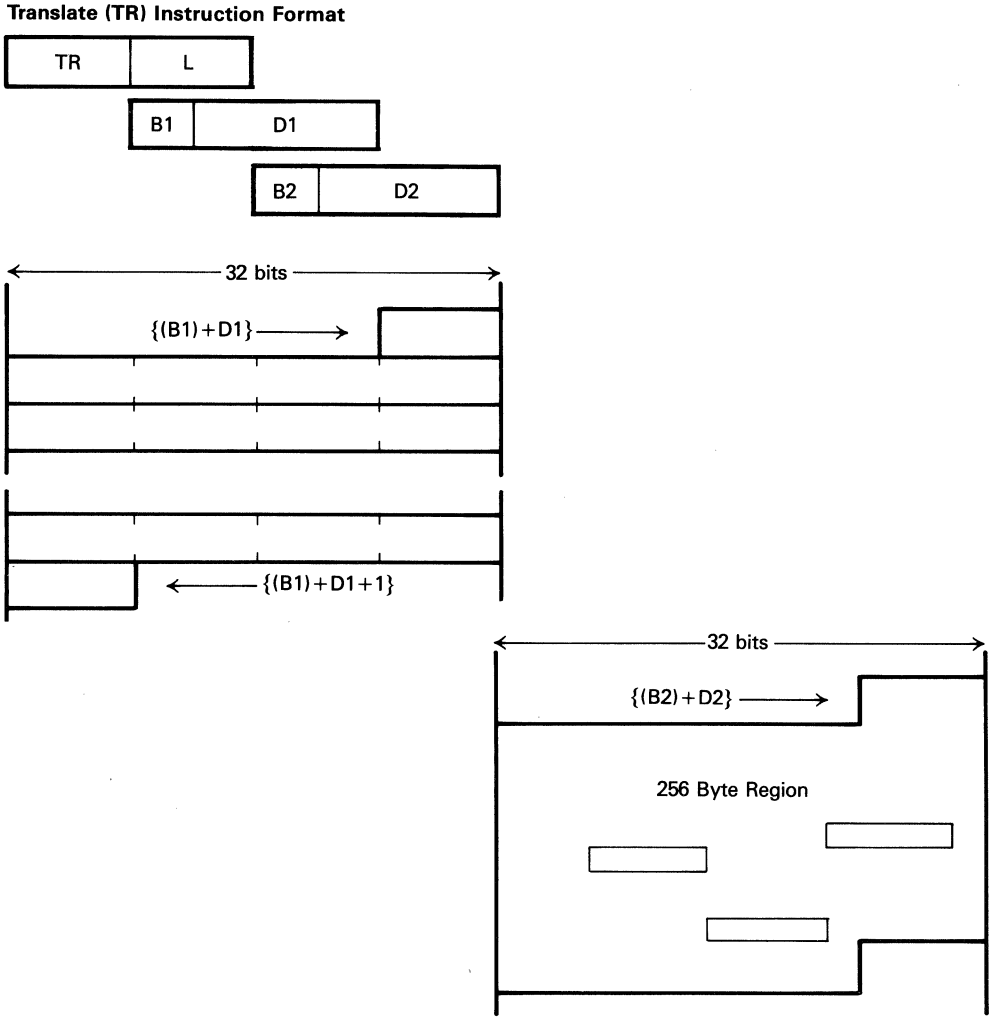


Figure 7.5 TR instruction format and memory operand format

field). You might just read (B2)+D2 and (B2)+D2+255. That's not legal. IBM System/370 architecture says the second operand byte does not have to be there if it does not participate in the execution of the instruction. You could search the first operand field for the smallest and the largest bytes and access the second operand field at (B2)+D2+smallest and (B2)+D2+largest. That would be okay, but it would be difficult. I chose the simpler method of practicing the instruction without changing any official state. I access all the bytes of both operands without changing

any state. If I can get them all, I then begin again and execute the instruction. Figure 7.6a, b shows flowchart sequences for addressing and execution.

Decimal Coprocessor Interface Microsequence

Micro/370 is designed optionally to work with a decimal coprocessor. The decimal coprocessor executes the nine System/370 decimal instructions. The instruction mnemonics are AP, CP, DP, ED, EDMK, MP, SP, SRP, and ZAP. These instructions operate on variable length decimal (base 10) operands and access these operands in memory. When the Micro/370 instruction decoder sees a decimal op code, it reads location 000000FA hex in real memory (real as defined by *IBM System/370 Principles of Operation*) and checks it. If it contains a zero, there is no decimal coprocessor. The instruction is treated as an operation exception.

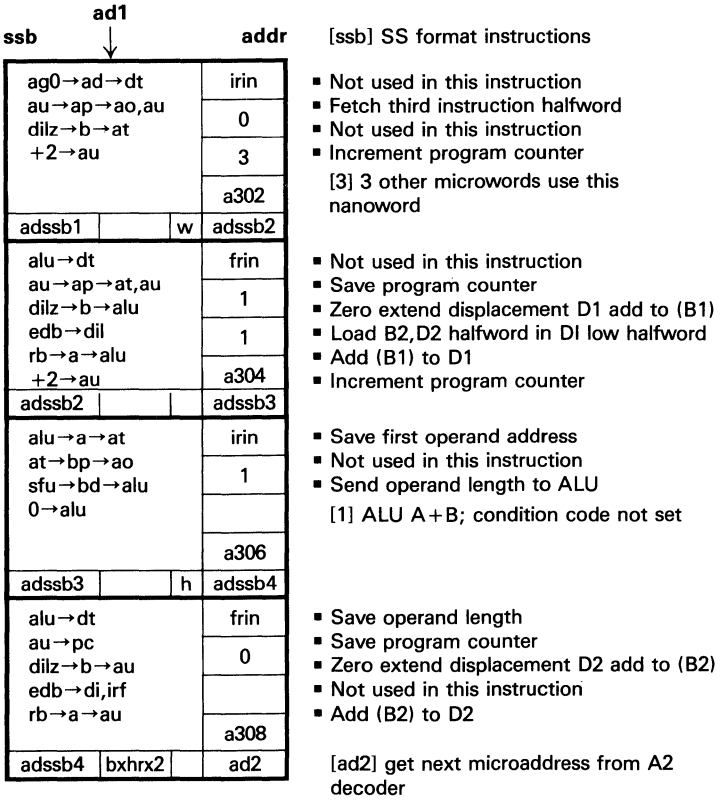
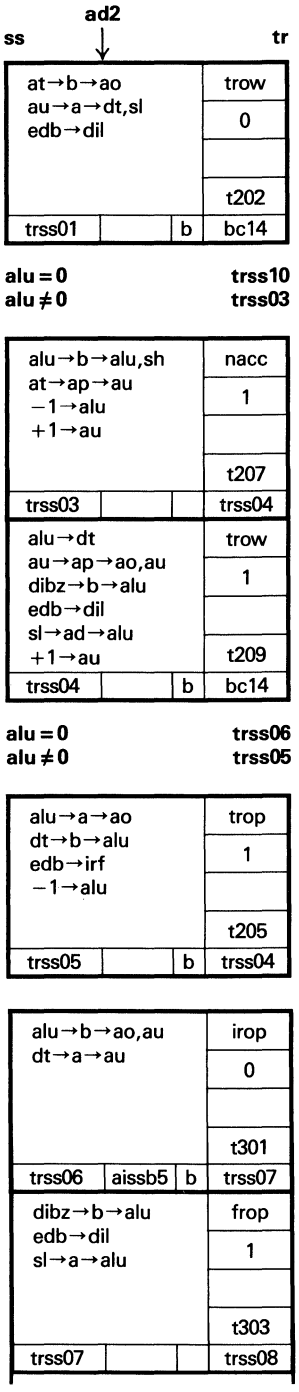


Figure 7.6a Flowchart sequence for TR operand addressing



[tr] translate instruction
[trow] total read from operand write space
- Read first byte of first operand
- Save address of operand two
- Load byte in DI low halfword

[bc14] branch condition 14 (checks for last byte of first operand)
Last byte of first operand
First operand longer than one byte

- Save operand length count
- Increment first operand address
- Decrement operand length count

- Save length count
- Read next first operand byte from read space
- Zero extend previous first operand byte
- Load next first operand byte in DI low halfword
- Add offset to second operand address

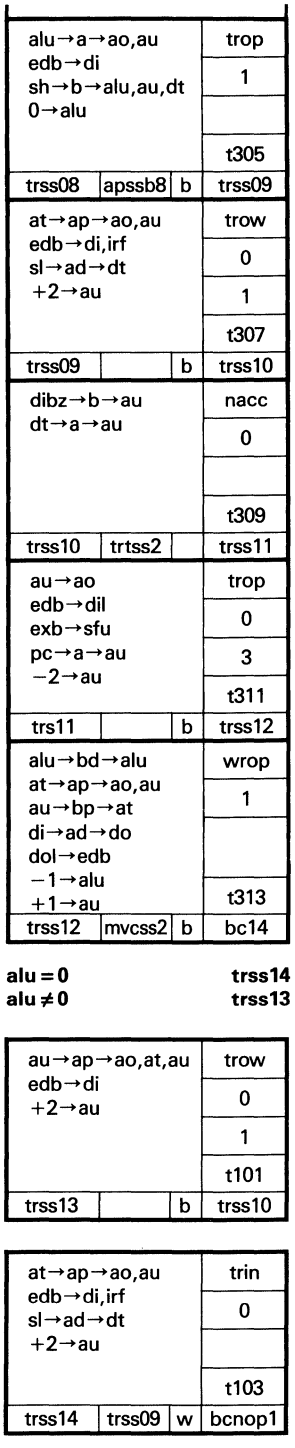
[bc14] branch condition 14 (tests condition set in trss03; checks for last byte of first operand)
Last byte of first operand
First operand longer

- Read second operand byte
- Decrement length count
- Not used in this instruction

[trss04] trss04 and trss05 form a tight loop to access all operand bytes that will participate in the instruction
[irop] initiate read of second operand
- Read second operand
- Not used in this instruction

[aissb5] origin of nanoword used by trss06
[frop] finish operand read
- Zero extend previous first operand byte
- Load next first operand byte in DI low halfword
- Add offset to second operand address

Figure 7.6b Flowchart sequence for TR execution



- Read next byte of second operand
- Load next second operand byte into DI
- Move operand length to ALU

- Read next first operand byte
- Load next first operand byte into DI
- Second operand base address (B2)+D2 to DT
- Not used in this instruction

- Add zero extended first operand index
- Add second operand base address

- Read next second operand byte
- Load next second operand byte in DI low halfword
- Not used in this instruction
- Prepare to refetch prefetch (after write) (IBM System/370 allows store-in-instruction stream)

[wrop] write operand
- Store byte in first operand location
- Save next instruction address
- Store translated byte
- Store from low order byte in DO buffer
- Decrement operand length count
- Increment first operand address

[bc14] branch condition14 (tests condition set in trss08 first, then previous trss12; checks for last byte of first operand)

alu = 0 **trss14** Last byte of first operand
alu ≠ 0 **trss13** First operand longer

- Read next first operand byte; save address
- Load next first operand byte in DI
- Not used in this instruction

[trss13] end of loop for translate

[trin] read next instruction halfword
- Refetch next instruction
- Load next instruction in IRF
- Not used in this instruction
- Increment program counter

[w] word read request to bus controller
[bcnop1] branch to NOP which does another halfword instruction fetch and loads IRD

Figure 7.6b (continued)

If location FA hex contains a nonzero number, Micro/370 multiplies it by 16, extends the sign, and stores the operation code and some flags at that address in service cycle memory. Micro/370 calculates the operand addresses and stores them at the base address, plus 16 and plus 32 in the service cycle memory. External hardware wakes up the decimal coprocessor.

Hardware attached to Micro/370 detects a service cycle write to the decimal coprocessor communication location base address. This hardware takes bus control away from Micro/370 and gives it to Micro/292 (the decimal coprocessor chip). Meanwhile, Micro/292 has been waiting for the response to its own service cycle read of the service cycle communication location since either power-up or the last decimal instruction. (On power-up and after each decimal instruction, Micro/292 initializes itself and then attempts to read the service cycle communication location. Micro/292 will be unable to complete the read cycle because it cannot get control of the external bus.) This happens at the end of state decm11 in, jumping ahead a little, figure 7.9. From the point of view of the flowcharts, the decimal coprocessor is transparent. The Micro/370 state sequencer continues to state decm12, which is a service cycle read to check the flags set by Micro/292. External bus control was given to Micro/292 during decm11, so Micro/370 "hangs" doing the service cycle read until Micro/292 and the external hardware return bus control to Micro/370.

External hardware translated Micro/370's last service cycle write into signals that took bus control away from Micro/370 and made Micro/292 the bus master. Micro/292 reads the operation code and flags from service cycle memory. The flags' information includes the mode and operand space selection bits currently being used by Micro/370. Micro/292 sets its bits to match these and then executes the decimal instruction.

Micro/292 is responsible for operand accesses, checking, instruction execution, and operand store. Upon completion of instruction execution, Micro/292 returns a set of flags to the service cycle memory communication location. External hardware detects this service cycle write and returns bus arbitration control to Micro/370. Micro/370 sets the condition codes (and the general register indicates the R1 field, if required) and completes the instruction. Exceptions detected by Micro/292 (access exception, specification exception, data exception, divide fault, and decimal overflow) are handled by passing enough information to

Micro/370 in flags so that Micro/370 can process the exception correctly. After writing the flags to service cycle memory, the decimal chip clears and resets its internal state to prepare for the next instruction.

Before Micro/292 can execute a decimal instruction, it must be initialized. To initialize Micro/292, Micro/370 must write the desired service cycle communication area address (divided by 16) in location FA hex in real memory, then reset Micro/292 by asserting a reset (CPU reset, initial CPU reset, or initial program load [IPL]) or the restart interruption to the decimal chip. If the restart interruption is used, Micro/292 will acknowledge the interruption; otherwise, the initialization follows the release of the asserted reset condition. Upon reset, Micro/292 reads location FA hex in real control memory, computes the address of the service cycle memory communication area, resets and initializes its internal registers, and attempts to read the communication location in service cycle memory. Since Micro/370 is the bus master, Micro/292 "hangs" until the external hardware detects Micro/370 writing to the decimal coprocessor communication location (in service cycle memory) and gives control to Micro/292. Figure 7.7 shows the decimal instruction format and service

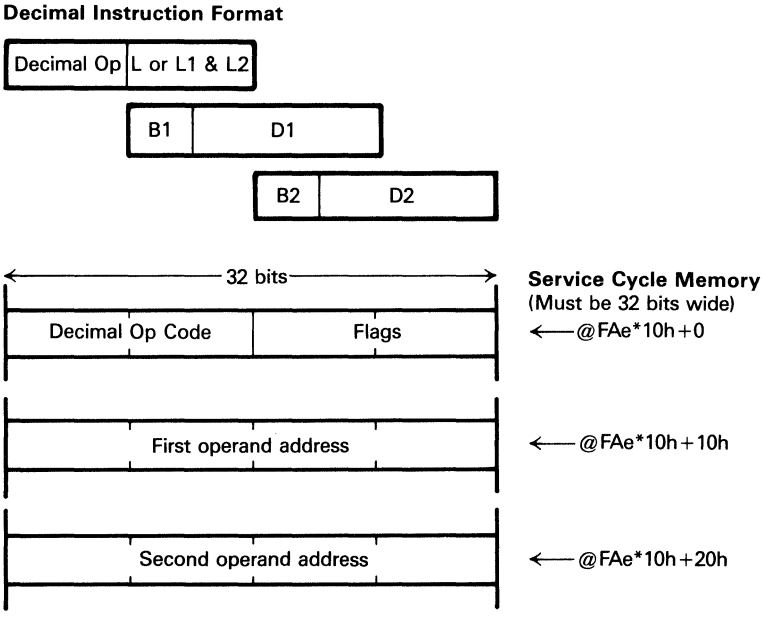


Figure 7.7 Decimal instructions and coprocessor communication in service cycle memory

cycle memory locations for coprocessor communication. Figure 7.8 is the flag format for coprocessor communication. Figure 7.9 (pages 173–176) is the Micro/370 flowchart sequence for decimal instructions.

As written by Micro/370 (at (@FAe*10h + 0))
Bits 0-15 are the first instruction halfword.

```
16   19  20   23  24   27  28   31
x x C C  x D x x  x x x M  R W x x
   | |      |         |     | | |
Condition  Decimal     Mode  | | 'Don't care'
code       overflow mask bit | Operand write space
                             Operand read space
```

As written by Decimal (at (@FAe*10h + 0))

Normal completion

```
16   19  20   23  24   27  28   31
0 0 0 0  0 0 0 R  x x x x  x x C C
|        |                   | | |
Must be  0→Do not replace R1   | Condition code
zero     1→Replace R1 with     'Don't care'
         value in (@FAe*10h + 10)
```

Access exception

```
1 x x x  x x x E  x x x x  x x x x
|        |
Must be  0→Error on read-only operand
one      1→Error in read/write operand
         (Byte address in (@FAe*10h + 10h))
```

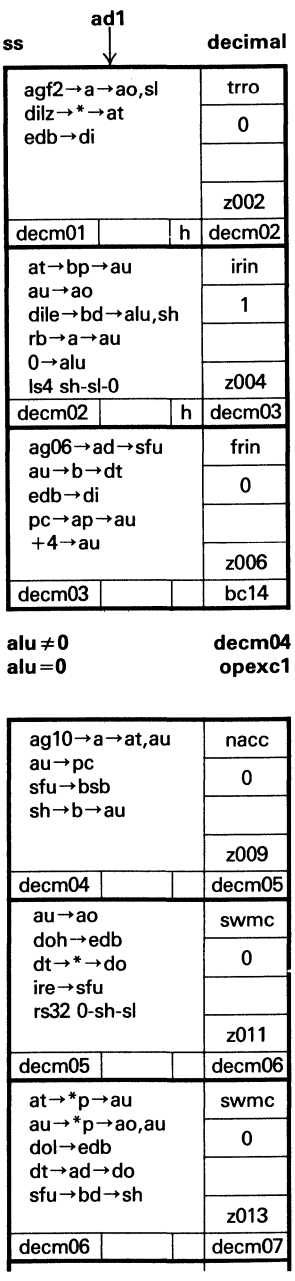
Program interruption

```
0 0 0 0  V V V V  x x x x  x x C C
```

	0 1 1 0	Specification	Suppressed
	0 1 1 1	Data	Suppressed or terminated
	1 0 1 0	Decimal overflow	Completed
	1 0 1 1	Decimal divide	Suppressed

Operand address 1 will be written in (@FAe*10h + 10h)
Operand address 2 will be written in (@FAe*10h + 20h)
All transactions take place in service cycle memory
R1 value passed in (@FAe*10h + 10h)
@FAe*10h is the sign-extended contents of FA times 10 (hex)

Figure 7.8 Decimal instruction flag word format



[ss] all decimal instructions are SS format
[decimal] all decimal op codes
[trro] total read in real memory

- Read from location 000000FA and save in SL
- Zero extend displacement D1 and save in AT
- Load contents of FA in DI

[h] halfword read
[irin] initiate instruction read

- Add displacement D1 to (B1)
- Read instruction halfword
- Sign extend contents of FA and save
- Add (B1) to displacement
- Test contents of FA for 0
- Multiply sign extended FA value by 4

[frin] finish read of instruction

- Load data strobe value in SFU
- Save first operand base address
- Load third instruction halfword in DI
- Increment the program counter

[bc14] branch condition 14/check for 0 in location FA

[alu≠0] coprocessor present
[opexc1] no coprocessor, instruction becomes an operation exception
[nacc] no bus cycle requested

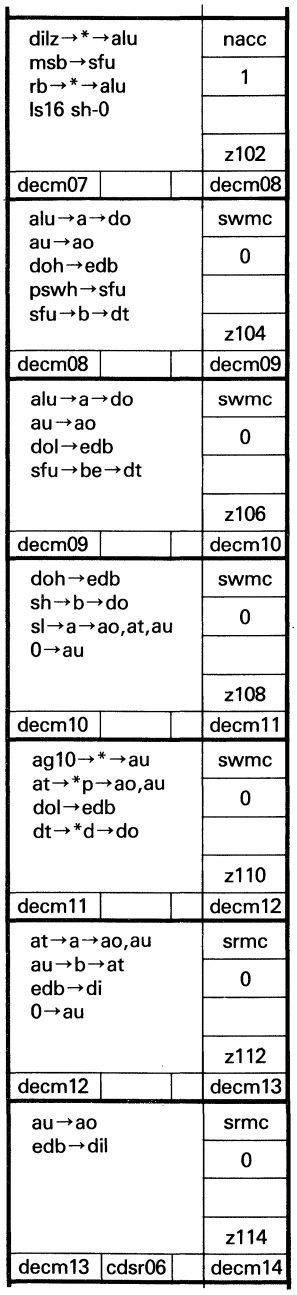
- Save constant 10
- Save program counter
- Load strobe values into strobe register
- Add 10 to @FAe*4

[swmc] service cycle write using bsb strobes

- Write to location @FAe*4+10h
- High-order halfword to external data bus
- Write first operand base address
- Move op code from IRE to SFU
- Move @FAe*4 address to SL

- Add 10 to current address @FAe*4+10h
- Write to service cycle memory
- Low-order halfword to external data bus
- Write first operand base address
- Send op code to shifter

Figure 7.9 Decimal instruction flowcharts (continues)



- Zero-extend displacement D2
- Mode status byte (etmssrr) to SFU
- Add (B2) to displacement D2
- Left shift op code 16 bits

- Write second operand address
- Write to @FAe*4+20h in service cycle memory
- High-order halfword to external data bus
- PSWH to SFU (get program mask bits and CCs)
- Save mode status byte (etmssrr) in DT

- Write second operand address
- Write to @FAe*4+20h in service cycle memory
- Low-order halfword to external data bus
- Move three high-order bytes of PSWH to DT

- High-order halfword to external data bus
- Send op code to external data bus
- Write to @FAe*4 in service cycle memory
- Save address in AT and AU

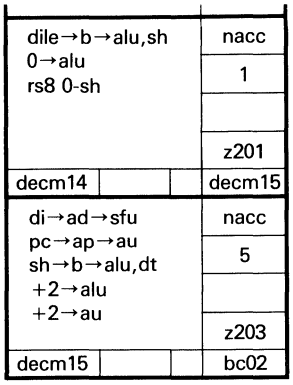
- Add 10 to address
- Write to @FAe*4 in service cycle memory
- Low-order halfword to external data bus
- Send flags to external data bus

[srmc] service cycle read using bsb strobes

- Read from @FAe*4 in service cycle memory
- Save incremented address @FAe*4+10h in AT
- Load flags in DI

- Read from @FAe*4 in service cycle memory
- Load flags in DI

Figure 7.9 (continued)



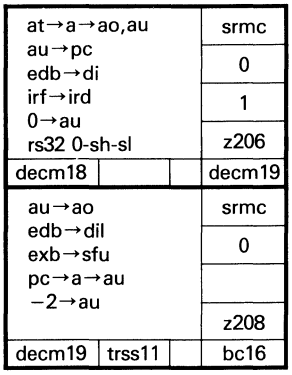
- Sign-extend flag halfword
- Test flag halfword
- Right shift sign-extended flag halfword

- Flag halfword to SFU (contains CCs)
- Update program counter
- Save flag halfword in DT
- AND flag halfword with 2

[5] AND A and B

[bc02] branch condition 2 (testing high-order flag bit)

alu<0 **decm18** [alu<0] decimal operand access exception
alu≥0 **decm16** [alu≥0] no access exception



- Read from @FAe*4+10h
- Save program counter
- Load access exception address high halfword
- Not used in this instruction

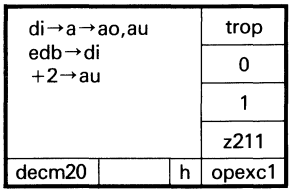
- Move high-order flag byte to SL

[1] 1 other microword uses this nanoword

- Read from @FAe*4+10h
- Load access exception address high halfword
- Not used in this instruction
- Decrement PC

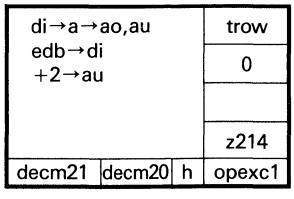
[trss11] first use of this nanoword
[bc16] branch condition 16/test read/write error flag bit

sl (31)=0 **decm20** [sl (31)=0] error on read-only operand
sl (31)=1 **decm21** [sl (31)=1] error on read/write operand



[trop] total read from operand read space

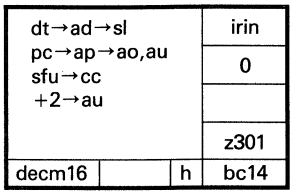
- Read from access exception address
- Not used (I expect an access exception)
- Increment operand address



[trow] total read from operand write space

- Read from access exception address
- Not used (I expect an access exception)
- Increment operand address

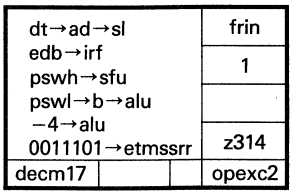
Figure 7.9 (continued)



[irin] initiate read of instruction
- Move high-order flag byte to SL
- Prefetch next instruction halfword
- Load result condition codes
- Increment program counter

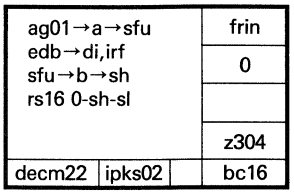
[bc14] branch condition 14/test for normal completion

alu=0 **decm22** [alu=0] normal completion
alu≠0 **decm17** [alu≠0] program interruption condition



[frin] finish read of instruction
- Send interruption code to SL
- Not used in this instruction
- PSWH to SFU
- Decrement PSWL to point to decimal op code
- Set mode, memory, and register control

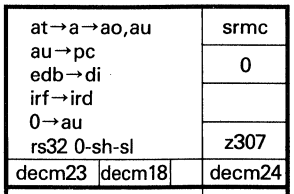
[opexc2] go to operation exception sequence



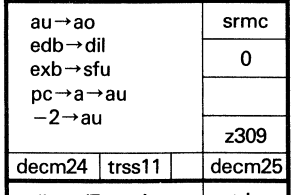
- Not used in this instruction
- Load next instruction halfword in IRF
- Not used in this instruction
- Not used in this instruction

[ipks02] original nanoword for this microword
[bc16] branch condition 16/check for Edit and Mark instruction

sl (31)=0 **bcnop1** [sl (31)=0] not Edit and Mark instruction
sl (31)=1 **decm23** [sl (31)=1] Edit and Mark instruction

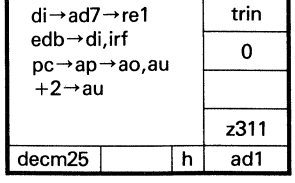


- Read from @FAe*4+10h
- Save program counter
- Load register R1 high halfword
- Load next instruction in IRD
- Move AT to AU
- Not used in this instruction



- Read from @FAe*4+10h
- Load register R1 low halfword
- Not used in this instruction
- Move program counter to AU
- Decrement program counter

[trss11] location of original nanoword
[trin] total read of an instruction



- Store the R1 value (three low-order bytes)
- Load prefetch instruction in DI and IRF
- Prefetch second following halfword
- Increment program counter

[ad1] get next microaddress from A1 decoder (decoded from IRD loaded in decm23)

Figure 7.9 (continued)

Convert to Binary (CVB)

Convert to Binary (CVB) changes an 8-byte packed decimal number from memory into a single 32-bit binary number in a general-purpose register (figure 7.10). The instruction checks the second operand (the memory operand) for valid sign and digit codes. Since the first operand (the result) is 32 bits (base 2), the largest (two's complement) number that can be represented is +2,147,483,647; the smallest is −2,147,483,648. The second operand is 64 bits and represents 15 digits and a sign in decimal. That's more than the 10-digit (equivalent) result. The instruction checks for overflow. If there is an overflow, the low-order 32 bits of the correct result are reported in the general register in the R1 field and a fixed-point divide exception is signaled.

How do you convert the packed decimal number to binary? Let's begin by ignoring the sign. Suppose you want to convert the decimal number 5,847 to binary. Here's one way to view 5,847:

$$5847 = 5*1000 + 8*100 + 4*10 + 7 = (((5*10 + 8)*10 + 4)*10) + 7$$

If you look at the number a digit at a time, you are not converting to binary but to hexadecimal (base 16). In base 16, all the decimal digits are valid, so you can convert the number by using the above formula and doing the adds and multiplies in base 16 instead of base 10. Begin with the high-order digit. Multiply by 10 (A in base 16), then add 8, and so on. Here it is:

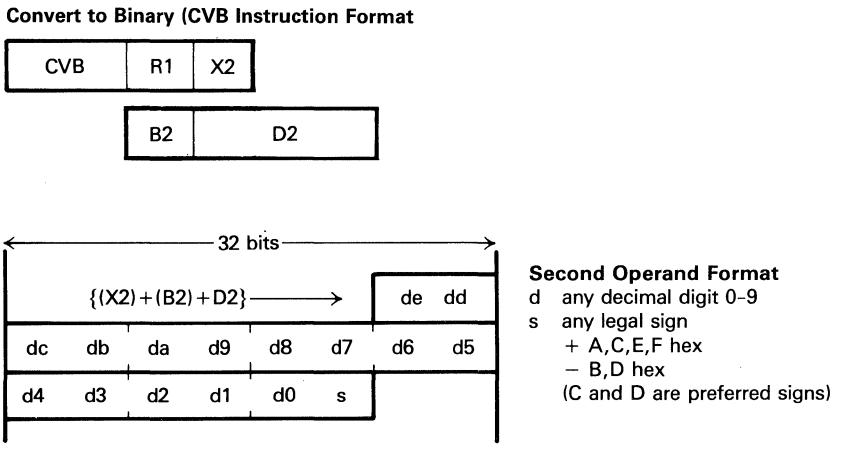


Figure 7.10 CVB instruction format and memory operand format

```
5*A = 32
      32 + 8 = 3A
            3A*A = 244
                  244 + 4 = 248
                        248*A = 16D0
                              16D0 + 7 = 16D7
```

16D7

Once you have the number, check the sign of the original number. If it was positive, the result stands. If the original number was negative, you need the two's complement of the result. For the above example, the negative result is FFFFE929 (in a 32-bit representation).

Now you have an algorithm. All you have to do is figure out how to do it with the available hardware. The adds are no problem. The result has to fit in a single 32-bit register, so if the result gets too big, you just record the overflow and throw out the overflow bits (keeping the low-order 32 bits). All that remains is to figure out how to multiply the base 16 partial result by A hex (10 decimal). There are many ways to do that. You could add the number to itself to get twice the number. Save that. Add the number to itself again—that's four times the number. Add that to itself yet again—that's eight times the number. Add the "eight times the number" to the "twice the number" you saved earlier, and you have ten times the number. You are ready to add the next digit. Another way to get ten times the number is to add it to itself to get two times the number and to shift the original number left by three to get eight times the number. Add those partial results for ten times the number. There are many ways to get ten times the partial result. Some are better than others. If you use a shifter to multiply by 2, 4, or 8, you must be able to detect overflow. If you use an ALU for all the operations, you have to check everything for overflow.

The CVB algorithm for Micro/370 reads the second operand (the memory operand) a byte at a time. It checks and converts two digits while it is prefetching the next data byte. I used the shifter as a 64-bit unit. I added the digit to the partial result, then I shifted left by two to multiply by 4. I added the unshifted sum to the shifted sum to get five times the original number. Then I shifted the number left again (multiplying by 2) to get the partial result plus the digit, times 10. I kept the partial result in SH

(shifter high) and shifted most of the overflow bits into SL (shifter low). The only overflows I had to check inside the conversion loop were adding the digit to the partial result and adding four times the result to the result. If an overflow occurred during either add, I put some nonzero character in SL. At the end of the algorithm, I checked SL for zero. If it was zero (I initialized it to zero), then there was no overflow. The last thing I did was check the sign and produce the two's complement result. (You have to check that for overflow, too.)

That was how I originally programmed the CVB instruction. Some time later (about two years), I discovered a way to check the decimal sign and digits a word at a time. I thought I could speed the algorithm, so I went back and studied it. I noticed that the result that could fit demanded at least five leading zeros in the second operand. I rewrote the algorithm to read and check two words and built a mask to check the leading digits. That left me with many fewer cases to check for overflow. (Only the last add could overflow.) I did not have to convert the first five digits (they all were supposed to be zero). The algorithm would be turbocharged. I wrote it. It was turbocharged. It also was wrong. That's how I found out the overflow cases had to give correct partial results. Back to the original algorithm. For the algorithms I chose, the architecture specification regarding overflow cases adversely affected the performance of the instruction, even for cases that did not overflow.

Convert to Decimal (CVD)

Convert to Decimal (CVD) is the opposite of CVB. CVD changes a 32-bit binary number in a general-purpose register into an 8-byte packed decimal number in memory. (See figure 7.11.) There is no chance for overflow. There are no valid sign and digit codes to check because there are no illegal bit patterns in the 32-bit first operand. The register designated by the R1 field is converted and stored in the second operand location.

Suppose you want to convert the hexadecimal number 3EB7 into decimal. Here's another way to view 3EB7:

```
3EB7 = 0011 1110 1011 0111
     = ((((((((((((((0*2+0)*2+1)*2+1)*2+1)*2+1)*2+1)*2+1)*2+0)
       *2+1)*2+0)*2+1)*2+1)*2+0)*2+1)*2+1)*2+1
```

This is the same situation I had with CVB. If you look at the binary representation of the number, you can represent all the digits in decimal. All you have to do is add the digits (ones and

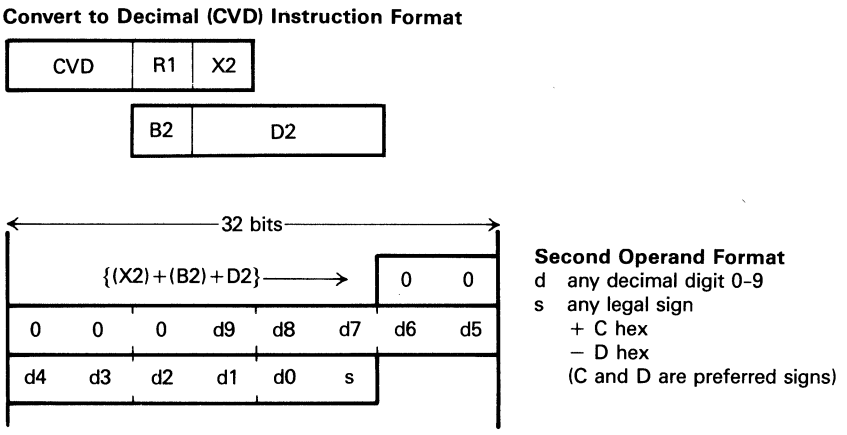


Figure 7.11 CVD instruction format and memory operand format

zeros) and multiply by 2 *in decimal,* and you will get the correct decimal result. Here it is:

```
      0
  0*2+0=0
    0*2+1=1
      1*2+1=3
        3*2+1=7
          7*2+1=15
            15*2+1=31
              31*2+0=62
                62*2+1=125
                  125*2+0=250
                    250*2+1=501
                      501*2+1=1003
                        1003*2+0=2006
                          2006*2+1=4013
                            4013*2+1=8027
                              8027*2+1=16055
```

16055

Again ignore the sign. If the beginning number is negative, attach the preferred minus sign to the result. If positive, attach the preferred plus. But what about converting negative numbers? Test for negative and complement if negative. The conversion is

(shifter high) and shifted most of the overflow bits into SL (shifter low). The only overflows I had to check inside the conversion loop were adding the digit to the partial result and adding four times the result to the result. If an overflow occurred during either add, I put some nonzero character in SL. At the end of the algorithm, I checked SL for zero. If it was zero (I initialized it to zero), then there was no overflow. The last thing I did was check the sign and produce the two's complement result. (You have to check that for overflow, too.)

That was how I originally programmed the CVB instruction. Some time later (about two years), I discovered a way to check the decimal sign and digits a word at a time. I thought I could speed the algorithm, so I went back and studied it. I noticed that the result that could fit demanded at least five leading zeros in the second operand. I rewrote the algorithm to read and check two words and built a mask to check the leading digits. That left me with many fewer cases to check for overflow. (Only the last add could overflow.) I did not have to convert the first five digits (they all were supposed to be zero). The algorithm would be turbocharged. I wrote it. It was turbocharged. It also was wrong. That's how I found out the overflow cases had to give correct partial results. Back to the original algorithm. For the algorithms I chose, the architecture specification regarding overflow cases adversely affected the performance of the instruction, even for cases that did not overflow.

Convert to Decimal (CVD)

Convert to Decimal (CVD) is the opposite of CVB. CVD changes a 32-bit binary number in a general-purpose register into an 8-byte packed decimal number in memory. (See figure 7.11.) There is no chance for overflow. There are no valid sign and digit codes to check because there are no illegal bit patterns in the 32-bit first operand. The register designated by the R1 field is converted and stored in the second operand location.

Suppose you want to convert the hexadecimal number 3EB7 into decimal. Here's another way to view 3EB7:

```
3EB7 = 0011 1110 1011 0111
     = ((((((((((((((0*2+0)*2+1)*2+1)*2+1)*2+1)*2+1)*2+1)*2+0)
       *2+1)*2+0)*2+1)*2+1)*2+0)*2+1)*2+1)*2+1
```

This is the same situation I had with CVB. If you look at the binary representation of the number, you can represent all the digits in decimal. All you have to do is add the digits (ones and

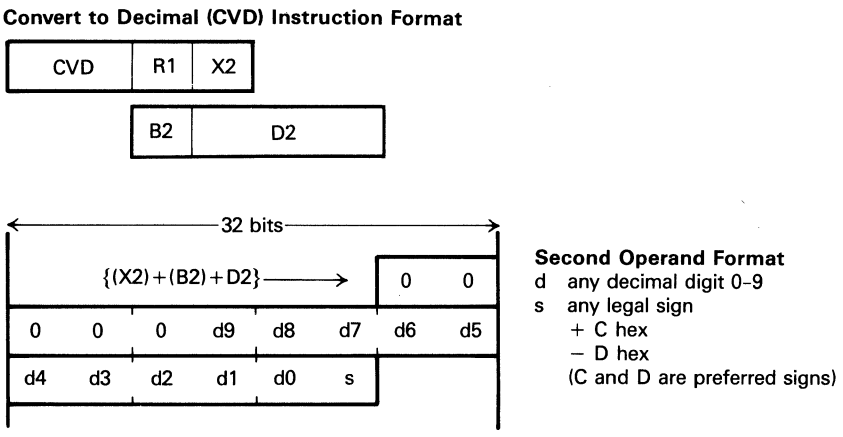


Figure 7.11 CVD instruction format and memory operand format

zeros) and multiply by 2 *in decimal,* and you will get the correct decimal result. Here it is:

```
    0
  0*2+0=0
    0*2+1=1
      1*2+1=3
        3*2+1=7
          7*2+1=15
            15*2+1=31
              31*2+0=62
                62*2+1=125
                  125*2+0=250
                    250*2+1=501
                      501*2+1=1003
                        1003*2+0=2006
                          2006*2+1=4013
                            4013*2+1=8027
                              8027*2+1=16055
```

16055

Again ignore the sign. If the beginning number is negative, attach the preferred minus sign to the result. If positive, attach the preferred plus. But what about converting negative numbers? Test for negative and complement if negative. The conversion is

then correct, even for the largest negative number. Suppose you try to convert the largest negative number in a 16-bit word. The number is 8,000 in hexadecimal. Check to see whether it is negative. If it is, take the two's complement to get 8,000 hex, then convert.

```
      1
  1*2+0=2
    2*2+0=4
      4*2+0=8
        8*2+0=16
          16*2+0=32
            32*2+0=64
              64*2+0=128
                128*2+0=256
                  256*2+0=512
                    512*2+0=1024
                      1024*2+0=2048
                        2048*2+0=4096
                          4096*2+0=8192
                            8192*2+0=16384
                              16384*2+0=32768
```

−32768

I appended the minus sign in the above result because the original number was negative (it did not come from the algorithm). The number of decimal digits in the result (counting the sign) can exceed the number of hexadecimal digits (eight), so the result is larger than 32 bits. This implies multiprecision arithmetic in an execution unit with 32-bit registers and arithmetic units. The adds and multiplies are done in decimal (not binary or hex).

Is there a good way to do decimal arithmetic with a binary ALU? Not really. There is a way, but not a good way. You can't just look at bits as if the number is binary because the bit positions aren't weighted correctly for that. In binary, you double the weight of each bit position from right to left. This is true no matter how many bits the number has. In decimal, the bits representing a digit are weighted the same as in binary (8,4,2,1), but each digit position is given a weight of only 10 (instead of 16). This means decimal arithmetic has to deal with digits or groups of digits and nothing smaller. Shifting left and right by one does

not multiply and divide decimal numbers by 2. Shifting by two and three does not work either. Shifting left and right by four multiplies and divides the number by 10. You can shift in multiples of a digit.

Here's another example. You want to add the following decimal numbers:

```
  1 2 7 5 3 8
  3 1 4 8 9 8
```

You do not have a decimal ALU, but you have a 32-bit binary ALU. Here's a way to add the numbers:

```
  1 2 7 5 3 8
  3 1 4 8 9 8   ADD the numbers (in hexadecimal).
        <       "<" indicates a (hex) carry at this digit.

  4 3 B D D 0
  6 6 6 6 6 6   ADD 6s.
    <<<

  A A 2 4 3 6   Uncorrected sum.
- 6 6 0 0 0 0   Subtract 6 from each digit with no carry out.
  4 4 2 4 3 6   Decimal result.
```

I had a problem with this method. I had to remember that the OR of the digit carries for the two ADDs in order to generate the appropriate correction factor (−66000). That was leading me to messy hardware. Then I noticed if I added the 6s first, there was no way to get a carry. The largest legal decimal digit in any number is a 9. Adding the 6s biases the number so all the carries propagate at once. Here is the simplified addition:

```
  1 2 7 5 3 8
  6 6 6 6 6 6   ADD 6s.
                No carries can propagate.

  7 8 D B 9 E
  3 1 4 8 9 8   ADD the numbers (in hexadecimal).
    <<<<        "<" indicates a (hex) carry at this digit.

  A A 2 4 3 6   Uncorrected sum. Set condition codes here.
- 6 6 0 0 0 0   Subtract 6 from each digit with no carry out.
  4 4 2 4 3 6   Decimal result.
```

Subtraction seems easier. You don't need the bias (the 6s). Just subtract and correct.

```
  5 9 6 3 4 0
- 3 0 8 6 3 1   Subtract the numbers (in hexadecimal).
  <<     <      "<" indicates no borrow at this digit.

  2 8 D D 0 F   Uncorrected sum. Set condition codes here.
- 0 0 6 6 0 6   Subtract 6 from each digit with no carry out.
  2 8 7 7 0 9   Decimal result.
```

Here's what happens when you subtract a big number from a little one:

```
  2 6 8 6 4 1
- 9 4 2 4 5 2   Subtract the numbers (in hexadecimal).
   <<<          Borrow out indicates negative result.

  9 2 6 1 E F   Uncorrected sum. Set condition codes here.
- 6 0 0 0 6 6   Subtract 6 from each digit with no carry out.

  3 2 6 1 8 9   Decimal result. Ten's complement → negative.
  6 7 3 8 1 0   Nine's complement of result.
          1     Add 1 to form ten's complement.
- 6 7 3 8 1 1   Sign and magnitude result.
```

Decimal subtraction seems easier than decimal addition because the bias (all 6s) is not needed. Subtraction is actually more work. For the addition, I assumed positive numbers, and for the subtraction, I ended with a ten's complement result, but I did what looked like a sign and magnitude subtraction. So I either have to convert the original subtrahend to a ten's complement number and perform the subtraction just like the addition, or I have to convert the result to a sign and magnitude number. In the example above, I formed the nine's complement in one line. I cannot do that directly with only binary hardware. Here is a method that works with the same number of steps for Add and Subtract:

```
  2 6 8 6 4 1
- 9 4 2 4 5 2   Subtract the numbers (in hexadecimal).
   <<<          Borrow out indicates negative result.

  9 2 6 1 E F   Uncorrected sum. Set condition codes here.
  0 6 6 6 0 0   Add 6 to each digit with no borrow.

  9 8 C 7 E F   Intermediate result.
  6 7 3 8 1 0   One's complement of result.
          1     Add 1 to form two's complement.
- 6 7 3 8 1 1   Sign and magnitude result.
```

If you get a borrow out of the high-order digit, the result is negative. Add the correction factor (the 0s and 6s) and complement the result. If there is no borrow out of the high-order digit, the result is positive. Subtract the correction factor to get the result. Now let's get back to CVD. I avoided the problems with ten's complements and decimal subtraction by working with unsigned numbers. I checked the first operand and made sure it was positive before I began the conversion. (I already showed that the maximum negative number, which complements to itself, gives the correct conversion result.) First, I set up loop counters. I have to convert a word one bit at a time—32 conversion loops. I initialize the result to zero. Then, I begin scanning bits exactly as in the conversion examples above. I do the adds and multiplies (in decimal) exactly as the decimal addition above. Since I only need to multiply by 2, I just add the number to itself. Because the binary number is 32 bits, the decimal result can be larger than a single 32-bit word. The decimal adds and multiplies have to be double-precision.

My original control word sequence skipped the leading zeros in operand 1 at one microcycle per zero. Once I encountered a one, I began to accumulate the result in a double-precision temporary register. Later, I noted that the partial result cannot overflow until the decimal result exceeds eight digits (99,999,999+−units). That corresponds to the hexadecimal number 5F5E100. The significance of that number is that it is not possible to overflow until you have converted a number larger than 4,000,000.

Suppose you are converting the largest negative number (see above example). The high-order bit will be one and all other bits zero. There are no cases worse than the largest negative number. All other numbers begin with zero (since I made them all positive). You scan the number from left to right. The decimal result will not overflow the single 32-bit register until you have shifted enough times to have converted the equivalent of 8,000,000 (hex). Thus for the first 27 iterations, you can do a (significantly faster) single-precision conversion and then do the double-precision conversion. The algorithm I wrote actually does a single-precision conversion until it detects a carry into the high-order temporary register and then begins the double-precision (64-bit) conversion.

Multiply

Multiply is an ordinary instruction. IBM System/370 architecture says the first operand is multiplied by the second operand and

the result is placed in the first operand location. The multiplier (second operand) and the multiplicand (first operand) are 32-bit signed integers. The result is 64 bits and occupies a register pair (the first operand must designate an even register). I use a signed single-bit Booth's algorithm to do the multiply. Figure 7.12 shows simplified hardware and the algorithm I use. The same hardware and algorithm work for unsigned multiply if N⊕V is replaced by zero. I could have made the algorithm about twice as fast if I had the ALU do shift and add (and shift and subtract) in one cycle. I thought it would slow the ALU and consequently the whole design, so I didn't ask. I should have, because circuit designers tell me it wouldn't cost much.

Figure 7.13 is an example of a signed multiply using the hardware and algorithm in figure 7.12. The algorithm works by shifting over zeros and ones:

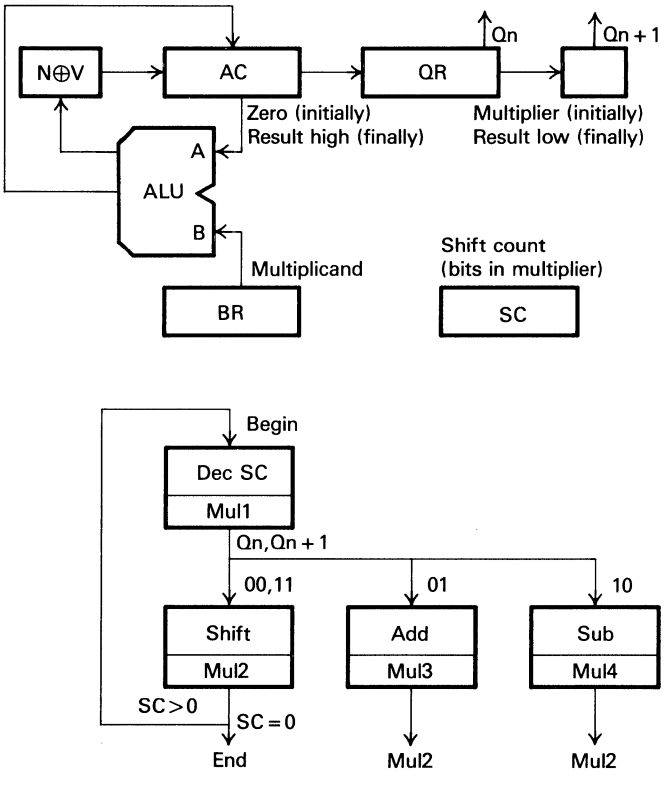


Figure 7.12 Hardware and algorithm for Booth's multiply

"... 0000 ..."	Shift past zeros. OK—just place holders. Multiply by 2.
"... 1111 ..."	Shift past ones. What's going on? Multiply by 2.
"....01...."	Add and shift. Must be at one end of a string of ones or zeros.
"....10...."	Subtract and shift. Must be at the other end of a string of ones or zeros.

For example, suppose you want to multiply X by 011110. The 1111 is treated as 10000–00001. (See figure 7.14.) The algorithm I used for Micro/370 looks at bit pairs and shifts 1 bit at a time. You can speed the algorithm by recognizing longer strings of ones and zeros and shifting over them. You can look at 3 bits at a time and shift over 2 bits at a time. Then you will add and subtract one or two times the multiplicand.

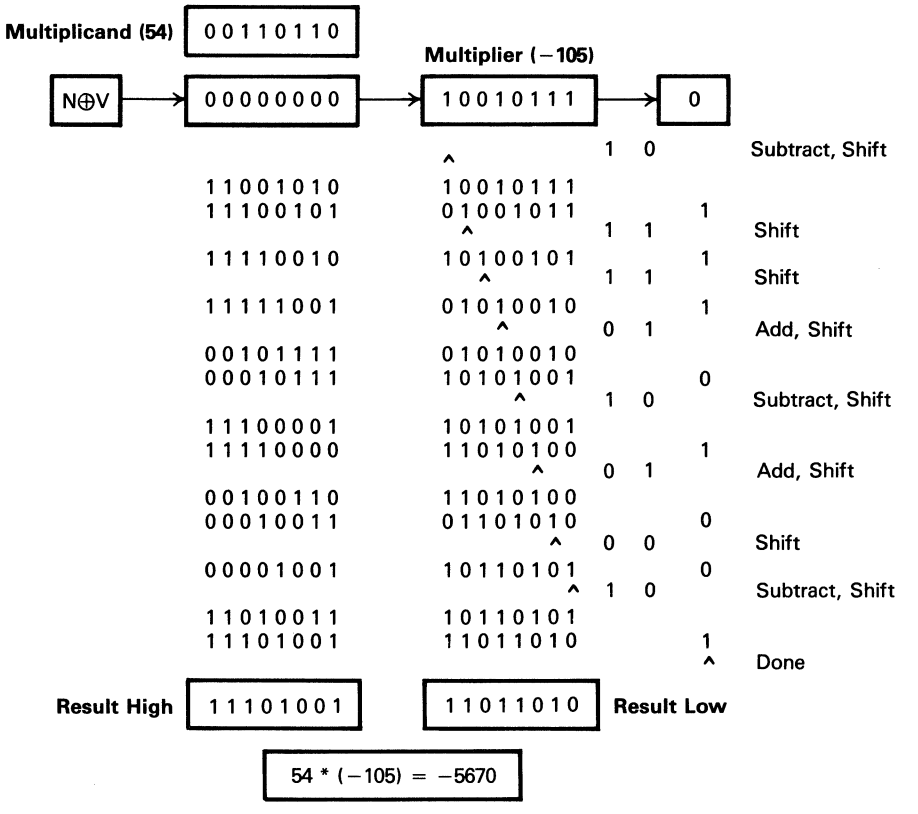


Figure 7.13 Example signed multiply

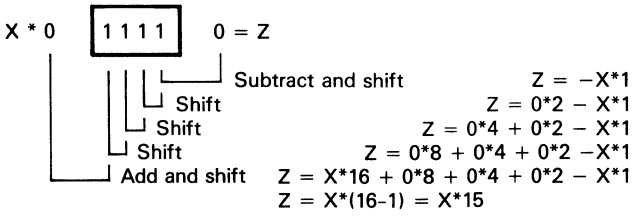


Figure 7.14 Single-bit Booth's algorithm multiply example

A shift operation always follows the add or subtract. It is possible to have the ALU do a shift-and-add or a shift-and-subtract. That would almost double the worst case performance of the algorithm (see figure 7.12). I should have asked for shift-and-add and shift-and-subtract. Because only a single bit shift is needed, it is not expensive. (This is another occasion where I didn't ask for something I thought would be too expensive. Ask for what you want and let the circuit designers tell you the cost. If they say it's too expensive, throw it out.)

Divide

The IBM System/370 Divide instruction specifies a 64-bit dividend in a general-purpose register pair and a 32-bit divisor. A 32-bit quotient and remainder replace the dividend in the register pair. I used a simple unsigned algorithm for divide for Micro/370. Just make both operands positive (really unsigned positive), then divide, then give the quotient and the remainder the appropriate sign. The hardware does the divide just as you would with a pencil and paper. Figure 7.15 shows an example of the algorithm and simplified hardware used for divide.

```
                  0 1 1 0 1   Quotient
              ---------------
1 1 0 1  ) 1 0 1 0 1 1 0 1   Dividend
Divisor       0
            -----
            1 0 1 0 1
              1 1 0 1
            -------
              1 0 0 0 1
                1 1 0 1
              ---------
                  1 0 0 0
                      0
                  -------
                  1 0 0 0 1
                    1 1 0 1
                  ---------
                      1 0 0   Remainder
```

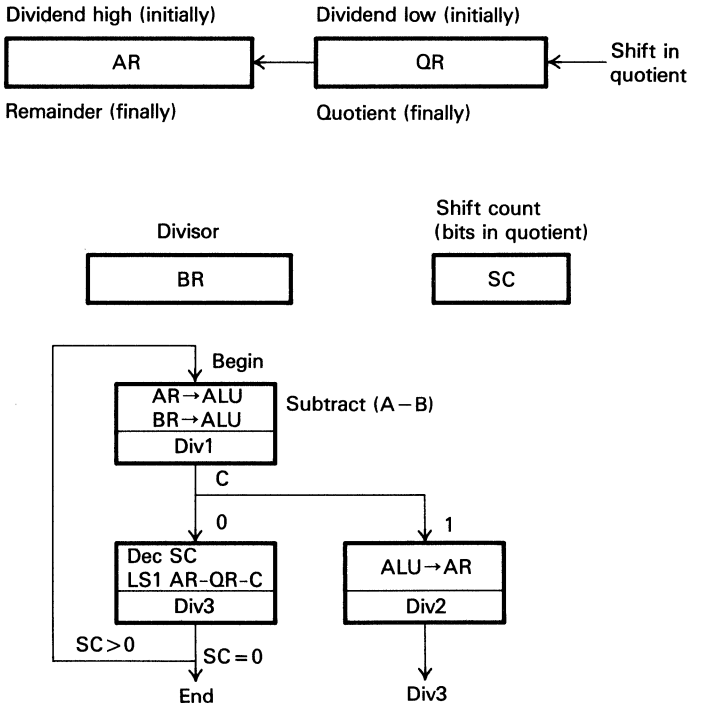


Figure 7.15 Hardware and algorithm for unsigned divide

The quotient is positive if the dividend and divisor are the same sign and negative if the signs differ. The remainder is given the sign of the dividend; a zero remainder is always positive. (The sign of the remainder is at the whim of the designer of the arithmetic. Sometimes the remainder is given the sign of the quotient.) Overflow occurs if the first subtraction produces a carry, which means that the value in AR is larger than or equal to the value in BR. If AR is larger than BR, the answer will be too large to fit only in QR. It is at this step, too, that divide-by-zero will be detected.

Divide is not as simple as I have made it sound. The divide loop is simple. Making the operands positive is easy (don't forget that the dividend is double-length). But you cannot just make the results the right sign and think you are done. The algorithm I showed is unsigned, so it can produce answers one bit larger

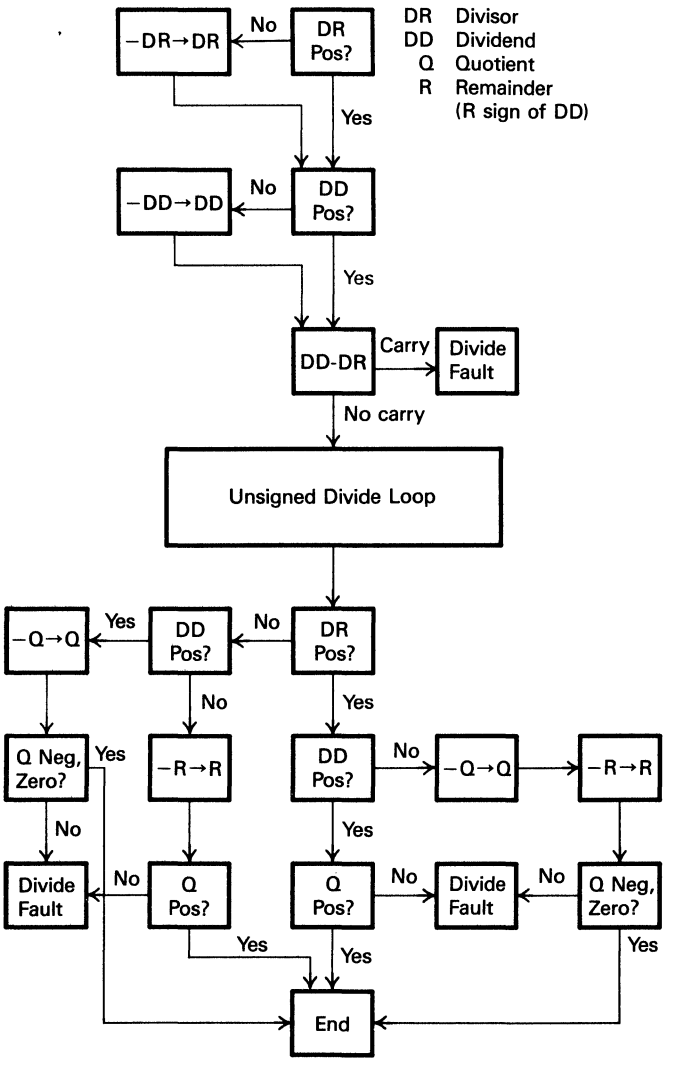


Figure 7.16 Algorithm for signed two's complement divide

than I can represent in two's complement notation (except for the largest negative number—another special case). You have to check the result to be sure it fits the result register. Figure 7.16 shows the algorithm for signed two's complement divide.

C H A P T E R

8

Implementing Micro/370 from Flowcharts

I started the Micro/370 flowcharts on 29 January 1981. When I finished them, I began the logic design. First I figured out the microword and nanoword formats. "What do you mean 'figured out'?" you might ask. "Aren't they part of the 'architecture' you start with? Doesn't somebody give you the block diagram of the micromachine, the microword format, and all that stuff?" Not with the flowchart method. It's not efficient to try to guess the block diagram and the microword format at the start of the design. The processor wouldn't do exactly what you wanted it to do. Don't worry about what the micromachine looks like or what the format of the control words is. Do the flowcharts and make the necessary (incremental) assumptions along the way. When you are done, use the flowcharts to collect the assumptions, then make those into the design. This chapter shows how I do that.

On 29 December 1981 (almost a year after I began working on them), I began typing the flowcharts into a computer file. Once I had the flowcharts in a file, I could look at them with a text editor. The editor helped me find all references to each execution unit macro. First, I listed the macros in the execution unit. Second, I searched the flowcharts, using the

editor to find all occurrences of the macro name. I listed the different uses of each macro under its name, then assigned each one bits sufficient for its control. If two macros were almost mutually exclusive (such as the shifter and the Pack-Unpack unit in Micro/370), I combined their control in a single field. In Micro/370, for example, shifter control needed 5 bits and Pack-Unpack control needed 4 bits. Since the Pack-Unpack unit and the shifter were seldom used in the same state, both macros could be controlled by a single 6-bit field. (See chapter 4 for a description of how to derive the nanoword format.) Figure 8.1 shows the final field assignments (field name and number of bits) for the nanoword and the associated uses from the flowcharts.

I began work defining the nanoword on 26 January 1982. On paper, I sorted the references according to macro. Figure 8.1 shows a sorted list of references for Micro/370. It is not the list I started with, because the execution unit has changed. The list shows the nanoword bit field and the number of bits I assigned. It also shows the task components from the flowcharts. Semicolons separate multiple references to the same unit in a single flowchart state. Multiple references must be treated as separate cases in the Karnaugh map for the decoder. There are 72 bits in the nanoword. I listed 71 (1 spare).

I collected references to the macros and assigned the number of bits to nanoword fields. I did not assign bit patterns to the nanoword fields yet because bit patterns are assigned as a result of the logic design procedure. This chapter shows how this is done for three of the nanoword fields. The circuit designers assign the control lines for each macro; the flowcharts give me the command set (the partial tasks listed for each field in figure 8.1); and the nanoword format tells me the number of bits representing the control (also shown in figure 8.1). I drew a Karnaugh map of the appropriate number of bits and filled in the commands listed under the name of the nanoword field (from figure 8.1). I moved the commands around in the Karnaugh map in an attempt to minimize implementation functions for the control lines. The final Karnaugh map contains the bit assignments for the nanoword field. I implemented the Karnaugh map with a PLA. Optimum placement of tasks in the Karnaugh map may result in a PLA with no OR array. (It may not be possible to place tasks in the Karnaugh map to achieve a PLA with no OR array.)

It took me about a year to complete the logic design for the nanoword field PLAs (I was working on flowcharts and other

AO--2 Bits
  ap→ao
  bp→ao
  au→ao
  none
EX--2 Bits
  ex→ap
  ex→bp
  bp→ex
  none
PSW--1 Bit
  pswl→bp
  none
PC--3 Bits
  au→pc
  ap→pc
  bp→pc
  ap→pc;pce→bp
  pce→ap
  pc→ap
  pc→bp
  none
AUA--1 Bit
  au→ap
  none
AUB--1 Bit
  au→bp
  none
AUI--4 Bits
  bp→au;ap→au
  bp→au;ap→au;s
  0→au;ap→au
  +1→au;ap→au
  +2→au;ap→au
  +3→au;ap→au
  +4→au;ap→au
  +6→au;ap→au
  +7→au;ap→au
  -1→au;ap→au
  -2→au;ap→au
  -3→au;ap→au
  -4→au;ap→au
  -7→au;ap→au
  -8→au;ap→au
  none
AT--3 Bits
  ap→at
  bp→at
  ap→at;at→bp
  bp→at;at→ap
  at→ap
  at→bp
  none
BUS--2 Bits
  ap=d
  bp=d
  ap^d
  bp^d
BMA--4 Bits

a2;b2
a7;bf
am;b1
af;bf
az;bf
af;b1
af;b7
af;be
af;bm
az;bz
am;bm
af;bz
a7;b7
AGB--1 Bit
  agxx→ad
  agxx→bd
AG--6 Bits
  ag0
  ag01
  ag02
  ag03
  ag06
  ag07
  ag08
  ag09
  ag0c
  ag0f
  ag10
  ag20
  ag28
  ag30
  ag40
  ag80
  ag86
  ag88
  ag8c
  agb8
  agba
  age8
  agf0
  agf2
  agfa
  agfe
  agff
  ag100
  ag3f00
  agff0f
  ag80000
  ag440000
  ag3700400
  ag66666666
  agb1000000
  agb2200000
  agb80840ff
  agff000000
  agfff00000
  agffff8000
  none
RA--4 Bits

ad→re1
ad→r2(1)
ad→r1(1)
ad→r2
ad→r1
rb→ad
x2→ad
r2→ad
r1→ad
r2(1)→ad
r1(1)→ad
none
RB--4 Bits
  bd→re2
  bd→r2(1)
  bd→r1(1)
  bd→r2
  bd→r1
  rb→bd
  x2→bd
  r2→bd
  r1→bd
  r2(1)→bd
  r1(1)→bd
  inc r1
  none
SPR--6 Bits
  1xxxxxx→etmssrr
  0xxxxxx→etmssrr
  x1xxxxx→etmssrr
  xxxxx01→etmssrr
  xxxxx00→etmssrr
  xxx00xx→etmssrr
  xxx01xx→etmssrr
  xxx10xx→etmssrr
  xxx11xx→etmssrr
  0x00011→etmssrr
  xxxxx11→etmssrr;ad→sl
  0011101→etmssrr;ad→sl
  ad→pur
  bd→pur
  pur→bd
  pur→ad
  ad→pur;pur→bd
  bd→pur;pur→ad
  bd→pur;ad→pur
  purh→ad;sh→bd
  purl→ad;sh→bd
  ad→sl
  bd→sh
  bd→sh1
  sh→bd
  sl→ad
  sh→bd;sl→ad
  ad→sl;bd→sh
  ad→sl;bd→sh;shift
  rs1 0-sh-sl;ad→sl
  rs1 0-sh-sl;bd→sh
  rs1 n.eor.v-sh-sl;bd→sh

rs8 0-sh;bd→sh
rs8 0-sh-sl;bd→sh
rs16 0-sh-sl
rs16 0-sh-sl;bd→sh
rs32 0-sh-sl
ls1 sh-0
ls1 sl-sh-0;bd→sh;ad→pur
ls1 sh-sl-0;bd→sh
ls1 sh-sl-1;bd→sh
ls1 sl-sh-0;bd→sh
ls2 sl-sh-0;bd→sh
ls4 sh-sl-0;bd→sh
ls8 sh-0
ls8 sh-0;bd→sh
ls8 sh-0;bd→sh1
ls8 sh-sl-0;ad→sl
ls16 sh→0
none
DTA--1 Bit
  dt→ad
  none
DTB--1 Bit
  dt→bd
  none
DTI--2 Bits
  alu→dt
  ad→dt
  bd→dt
  none
ALO--3 Bits
  alu→ad
  alu→bd
  corf→ad
  corf→ad;alu→bd
  alu→ad;alu→bd
  none
ACC--3 Bits
  1
  2
  3
  4
  5
  6
  0
  none
ALL--1 Bit
  bd→alu
  none
ALI--4 Bits
  ad→alu
  0→alu
  -1→alu
  -2→alu
  -3→alu
  -4→alu
  -6→alu
  -7→alu
  -8→alu
  +1→alu

+2→alu
+3→alu
+4→alu
+6→alu
+7→alu
none
SFU--5 Bits
  ad→sfu
  sfu→bd
  ad→sfut;sfu→bd
  ad→sfu;sfu→bd
  ad→sfut
  pswh→sfu
  sfu→pswh
  pswh→sfu;sfu→bd
  sfu→ilc
  ilc→sfu
  ilc→sfu;sfu→bd
  l1→sfu;sfu→bd
  bc→sfu;ad→sfu
  bsb→sfu
  sfu→bsb
  sfu→ird
  sfu→ire
  sfu→cc
  ire→sfu
  lvl→sfu
  ipb→sfu
  mim→sfu
  msb→sfu
  exb→sfu
  sfu→mim
  none
EDB--4 Bits
  edb→di
  edb→dil
  edb→irf
  edb→di;edb→irf
  edb→di;irf→ird
  edb→dil;irf→ird
  edb→di;edb→irf;irf→ird
  irf→ird
  ird→ire
  ad→do;dol→edb
  ad→do;doh→edb
  bd→do;dol→edb
  bd→do;doh→edb
  none
DI--3 Bits
  di→ad
  di→bd
  di→ad;di→bd
  dilz→ad
  dilz→bd
  dile→bd
  dibz→bd
  none

spare-1 Bit

Figure 8.1 Sorted list of nanoword references

things at the same time). At the end of that year, I had derived the nanoword bit patterns for each field.

I began defining the microword on 11 March 1982. The microword is 18 bits. It has a direct address format and a branch format. These formats are shown in figure 8.2. The first bit in the microword distinguishes the format. In the direct and branch formats, 5 bits are for function codes. The function code bits tell the bus controller what kind of external bus cycle the state sequencer is requesting.

Some of the microword bits become the external read/write line and the external function codes. The next microaddress field is the location of the next microword in the control word sequence. The type field tells the state sequencer where to get the next address. Usually it comes from the microword, but at the end of the address calculation sequence or the instruction sequence, the address comes from the instruction decoder.

In the branch format, the next address field is only 8 bits. The other 2 next microaddress bits come from the branch control unit. The branch condition field tells the branch control unit which branch condition selects the next address. There are thirty-one branch condition choices, requiring a 5-bit specification field. The function codes take 5 bits; the format (branch format or direct format) takes 1 bit; and the next address needs 8 bits (the other 2 bits are the output of the branch control unit). That's 19 bits, but there are only 18 bits in the format. If I added an-

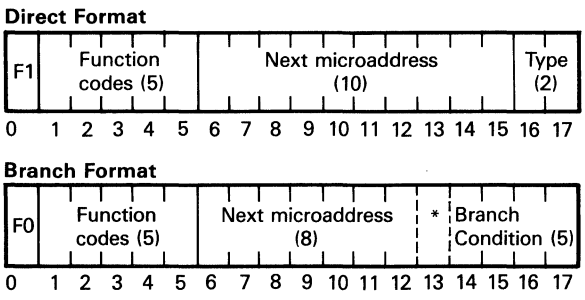

Direct Format
F1
Function codes (5)
Next microaddress (10)
Type (2)
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
Branch Format
F0
Function codes (5)
Next microaddress (8)
*
Branch Condition (5)
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
*Bit 13 is shared by next microaddress and the branch condition in the branch format.

Figure 8.2 Micro/370 microword formats

other bit to the branch format, all the microwords would become 19 bits. The control store would have to be larger.

At the time I worked on the microword format, I assumed the design would stack the microword control store on top of the nanoword control store so they could share common addressing. Stacking would be facilitated if the microword store and nanoword store could be made the same width. (The Motorola MC68000 has a rectangular control store. The microword portion has sixteen 17-bit control words per row. The nanoword portion has four 68-bit control words per row. Each row in the control store is 272 bits, whether it is nanowords or microwords.) If the microword is 18 bits and the nanoword is 72 bits, I can put sixteen microwords per row and four nanowords per row in the control store in a convenient rectangular layout. If the microword became 19 bits, the nanoword would become 76 bits—which was 5 bits more than I needed.

One bit in the microword format is shared between the branch specification field and the next control store address field. That means when the branch microwords are placed in the control store, one of the bits in the address is fixed by the specified branch condition. The high-order bit of the branch condition specification becomes the low-order bit of the next control store base address. (This control store base address is still supplemented by 2 bits from the branch control unit.) Sharing a bit in the branch format kept the microword at 18 bits. (See figure 8.2.)

Implementation

I will present three examples to show how to implement the execution unit controls from the flowchart description. The first example is the simple Address Out (AO) control. The second and third examples are the Pack-Unpack Register (PUR) control and the Special Function Unit (SFU) control, respectively. They are more complicated and more representative. The general design procedure is to extract the design specification from the flowchart description. The flowcharts contain the assumptions made for each functional unit. The design procedure produces a technology-dependent logic description. This description is suitable for implementation by circuit designers.

Address Output Example

The AO buffer is an amplifier (and latch) that holds the address for presentation at the Micro/370 pads. The flowcharts show only three uses of AO. An address can be transferred to AO from either the A or B internal bus or directly from the arithmetic unit (AU). I also must represent the case for no transfer (because there are flowchart states that do not use AO), so there are four uses of AO. I need assign only 2 bits in the nanoword for control of AO. (This is unusual. More typically, the perversity of nature makes number of cases some power of two plus one—e.g., 5, 17, 33.) I designed logic decode for AO on 11 February 1983 and revised it only three times by 2 May 1983. It is shown in figure 8.3, which is nearly a direct copy of the computer file for the PLA. N-AO is the 2 input bits from the nanoword AO field. The output control lines are the lines with names (ao03p2, for example, which means that it is the third AO control line and it is a phase 2 signal). The column labeled TRANSFER contains the line controls. In the case of line 3, 32 bits of the AU (au(0-31)) go to address output (auo(0-31)). The column labeled EQUATION usually shows the (partial) task(s) from the flowcharts for which the line will be active.

Just to show that nothing is simple in an industrial design, the AO is now on revision 8 (by 25 April 1984). It was modified (revision 7) to add control for the real address line ao(32), which comes from outside the execution unit. Later, control for the real address line was moved again.

```
* ORG 11 FEB 83 / 8 REV 25 APR 84 (RWH)
* FILE 'MFAO SCRIPT'
*
*
*                AO  CONTROL
*
*LINE NAME     N-AO      TRANSFER                EQUATION

3  ao03p2      11        auo(0-31)→ao(0-31)      au→ao

1  ao01p2      01        ap(0-31)→ao(0-31)       a→ao

2  ao02p2      10        bp(0-31)→ao(0-31)       b→ao
```

Figure 8.3 PLA text file for AO

I call the circuit in figure 8.3 a PLA, but it doesn't look like a PLA because it doesn't have an OR array. I call it a PLA because that's what I set out to design. If I can design my PLA well enough, then I will end up with a degenerate OR array.

Usually the circuit designers will ask me for a separate control line for each unique transfer in the execution unit. There are three possible sources for AO, and I need a separate control line for each source. Figure 8.4 shows separate true and complement lines coming from the nanoword control field (N-AO). Where I put a one in the array, there is a transistor on the complement line. Where I put a zero in the array, there is a transistor on the true line. An *x* means there is no connection. This is the AND array. Its inputs are the lines from the N-AO field. Its outputs are the lines at the left having a name and a number. Pull-up resistors at the bottom of the vertical lines try to bring all the columns of the AND array to a logic 1. The lines are encoded to tell you what nanoword value is required to activate the control line. In figure 8.4, for example, line ao02p2 is active (logic 1) when the nanoword field contains 10. If the nanoword N-AO field contains 00, then all the lines of the AO control PLA will be pulled low. Pull-up resistors on the right side of the array try to bring all the rows of the PLA to a logic 1. The ones and zeros in the AND columns show the existence of a transistor, driven by the vertical line, that pulls the corresponding horizontal line to ground if the vertical line is a logic 1. In this simple example, there is no OR array, so the horizontal lines are the control lines for the execution unit transfers.

Pack-Unpack Register Example

The PUR is a register with special properties to support execution of several System/370 instructions. It first appeared in the execution unit on 3 September 1981. I added it for the Pack instruction. PUR appeared in the flowcharts on 1 October 1981 in the Pack instruction.

I used PUR for other instructions too: MVN and MVZ (1 October 1981), UNPK (2 October 1981), MVO (2 October 1981), and CVB (11 November 1981). Each time I used PUR in the flowcharts, I made some assumption about what it would do for me for that particular instruction. In the Pack instruction, I assumed that if the operand came from the A bus, the high-order nibble of the low-order byte was stored in the high-order nibble of the (1-byte) PUR. If the operand came from the B bus, I assumed the low-

order nibble of the low-order byte was stored in the low-order nibble of PUR, and I assumed zero was stored in the high-order nibble. These transfers support the Pack instruction directly. Analogously, for other instructions, I assumed PUR did whatever was needed.

I began to implement the PUR control logic on 11 June 1982. The first drawing is dated 17 June 1982. Here is what I did:

1. I used a text editor to find all tasks using PUR as a source or destination. I copied the task and the instruction. The set of different uses of PUR determines how many bits must be assigned to the control field in the nanoword. There are six different uses (plus "none," which makes seven), so at least 3 bits must be assigned in the nanoword. (In this case, the nanoword field is shared between PUR and the shifter, so I assigned 5 bits on 10 February 1982.)
2. Under each different use for PUR, I listed the instruction where the task appeared and the function (how PUR manipulated its operands) I had assumed for that instruction. (Several instructions can share a transfer.)
3. I designed logic to support the transfers.
4. I made the first drawing of logic decode for PUR, dated 17 June 1982.

How did it turn out? All wrong. The final version doesn't look anything like the first drawing. But it was a start. I don't think I ever designed a decoder of more than a few lines that never changed. What went wrong? Nothing. Everything was proceeding normally. Requirements changed, my assumptions were wrong (about what I wanted, about what the circuit designers needed, or whatever), or (much more frequently) I just made

```
    AO  CONTROL

                NNNN
                AAAA
                0000
LINE    NAME    0011

3       ao03p2  1x1x

1       ao01p2  x01x

2       ao02p2  1xx0
```

Figure 8.4 Logic decode for AO

mistakes. I expect lots of revisions in a real design. No matter how carefully I design, no matter how many times I go over what I do, it never seems to be correct the first time. Educational design examples are different in that they lack the tedious extremes of checking, detail, verification, and redesign that are characteristic of every industrial design.

How did I find out it was all wrong? I showed it to the circuit designers, and they told me that what I had was no good for the technology—too many line crossings, too much random logic, unpredictable bidirectional data paths, and so on. I asked what they wanted. (Notes from the conversation are dated 1 September 1982—the original design lay dormant while we did other things.) They wanted a separate control line for each unique transfer. That changed the design procedure. Here is the design procedure for the design dated 1 September 1982:

1. I used a text editor to find all occurrences of PUR in the flowcharts.
2. Under each of the different uses for PUR, I listed the instruction where the task appeared and the function (how PUR manipulated the operands) I had assumed for that instruction.
3. I ordered the list according to the transfers. Figure 8.5 is the list of transfers specified by the flowcharts for PUR. It is ordered to correspond to the control lines for PUR in the execution unit. The order of control lines for the execution unit was determined by the circuit designers.

LINE	NAME	TRANSFER	EQUATION
9	sp06p1	a(24-27)-1	(pur→a)unpk
8	sp05p1	a(24-27)-0	((purh→a)+(purl→a))
7	sp04p1	a(0-23)-0	((pur→a)+(purh→a)+(purl→a))
4	sp02p3	a(28-31)-pur(4-7)	((a→pur)(unpk+cvb))
3	sp02p1	a(28-31)-pur(4-7)	((pur→a)+(purl→a))
10	sp07p3	0-pur(0-3)	(b→pur)pack
6	sp03p3	a(28-31)-pur(0-3)	(a→pur)(mvo+pack)
5	sp03p1	a(28-31)-pur(0-3)	purh→a
1	sp01p1	a(24-27)-pur(0-3)	(pur→a)$\overline{\text{unpk}}$
2	sp01p3	a(24-27)-pur(0-3)	(a→pur)$\overline{(\text{mvo+pack})}$
15	sp11p1	b(24-27)-pur(0-3)	(pur→b)$\overline{\text{unpk}}$
11	sp08p1	b(28-31)-pur(4-7)	(pur→b)$\overline{\text{unpk}}$
12	sp08p3	b(28-31)-pur(4-7)	b→pur
14	sp10p1	b(24-27)-1; b(28-31)-pur(0-3)	(pur→b)unpk
13	sp09p1	b(0-23)-0	pur→b

Figure 8.5 Execution unit control lines and transfers for PUR

	00	01	11	10
00	0 a→pur	1 sh→b purl→a	3 sh→b purh→a	2 X
01	4 a→pur pur→b	5 pur→b	7 a→sl	6 X
11	C b→pur a→pur	D b→pur	F a→sl b→sh shift	E b→sh a→pur ls1 sl←sh←0
10	8 X	9 b→pur pur→a	B pur→a	A X

X don't care

N-SPR	Flowchart Task
01 0000	a→pur
01 1101	b→pur
01 0101	pur→b
01 1011	pur→a
01 0100	a→pur;pur→b
01 1001	b→pur;pur→a
01 1100	b→pur;a→pur
01 0011	sh→b;purh→a
01 0001	sh→b;purl→a
01 1110	b→sh;a→pur
00 0000	none

Figure 8.6 Nanoword bit assignments for PUR control

4. I used a Karnaugh map to assign the nanoword bit patterns for PUR control. The Karnaugh map (only the part for PUR) and the final assignments for the bit patterns in the nanoword are shown in figure 8.6. PUR is controlled by a 6-bit N-SPR field from the nanoword. The N-SPR field also controls the shifter and the Mode Bits register.
5. I designed logic to support the transfers.
6. I made the second drawing of logic decode for PUR. It is dated 1 September 1982.

This, too, was all wrong, but the problems were subtler. The circuit designer did not want to implement the OR gate I used in the drawing, and he did not like the assumed dotted-OR connections. I tried to convince him that it was only a logical representation, but he was having none of that. I went through seven revisions of this sort. Further, when the circuit designers implemented the register and transfer paths in the execution unit, they found more problems. I had another discussion with Mon Yen Tsai (the lead circuit designer at the time) on 7 February 1983. He wanted a representation he could read from a computer data base and convert with a program to the final representation. On 7 February 1983, I threw out the seventh

revision of the original design and repeated the six design steps listed above. I had the first machine-readable description on 10 February 1983. The thirteenth revision of that description of PUR control logic is dated 9 May 1983. This was the final version. The latest version of PUR is shown in figure 8.7.

The final logic PLA for PUR, shown in figure 8.7, has 33 columns and 21 rows for a logic area of 693 units. This represents a PLA almost like the one implemented for Micro/370 (which looks to me to be about 644 units). In the Micro/370 PLA, the inhibit rows (lines 16, 17, 18, 19, 30, and 31 in figure 8.7) are placed among the control lines (lines 1 through 15) closer to the lines they inhibit, so there are only 4 inhibit columns.

This PLA has no OR array. It was placed on the Micro/370 chip on its side. The Shifter and Pack-Unpack Register (SPR) and IRE input lines enter from the right. The inhibits are at the bottom, and the control lines exit at the bottom. The PLA is shown in figure 6.1. The execution unit occupies the lower quarter of the chip. The largest solid-looking rectangle in the execution unit is the register file. PUR is just to the right of the register file. PUR is mostly a blank column with a byte of hardware at the bottom of the execution unit (it operates only on low-order bytes and nibbles). The small rectangle just above the execution unit is the PUR control PLA. (It is between a triangular register control PLA and a triangular shifter control PLA.)

Conventional PLA PUR Implementation

The way I implement a PLA is unusual. I do not know whether it is good or bad, but it works for me. Some people may not consider my units real PLAs because they are unconventional. They have inhibit lines that conventional PLAs do not, and often they do not have any OR array. I will take you through an exercise to find a conventional, folded, PLA implementation of the PUR PLA and then compare it with mine.

Figure 8.8 is a preliminary design for a conventional (meaning it has an OR array and no inhibit lines) PLA. Question marks in the rows under IRE bits indicate task equations for which there is no simple IRE encoding. These rows must be duplicated sufficiently to express the instructions with IRE bits (for instance, you must have enough lines to represent $\overline{\text{UNPK}}$). The conventional PLA for PUR is shown in figure 8.9.

PACK-UNPACK TRANSFERS AND EQUATIONS

NAME	TRANSFER	EQUATION
sp06p1	a(24-27)-1	(pur→a)unpk
sp05p1	a(24-27)-0	(purh→a)+(purl→a)
sp04p1	a(0-23)-0	(pur→a)+(purh→a)+(purl→a)
sp02p3	a(28-31)-pur(4-7)	(a→pur)(unpk+cvb)
sp02p1	a(28-31)-pur(4-7)	(pur→a)+(purl→a)
sp07p3	0-pur(0-3)	(b→pur)pack
sp03p3	a(28-31)-pur(0-3)	(a→pur)(mvo+pack)
sp03p1	a(28-31)-pur(0-3)	purh→a
sp01p1	a(24-27)-pur(0-3)	(pur→a)$\overline{\text{unpk}}$
sp01p3	a(24-27)-pur(0-3)	(a→pur)$\overline{(\text{mvo+pack})}$
sp11p1	b(24-27)-pur(0-3)	(pur→b)$\overline{\text{unpk}}$
sp08p1	b(28-31)-pur(4-7)	(pur→b)$\overline{\text{unpk}}$
sp08p3	b(28-31)-pur(4-7)	b→pur
sp10p1	b(24-27)-1;b(28-31)-pur(0-3)	(pur→b)unpk
sp09p1	b(0-23)-0	pur→b

PACK-UNPACK PLA ALMOST AS IMPLEMENTED IN MICRO/370

LINE	NAME	SSSSSSSSSSSS PPPPPPPPPPPP RRRRRRRRRRRR $0\bar{0}1\bar{1}2\bar{2}3\bar{3}4\bar{4}5\bar{5}$	IIIIIIII RRRRRRRR EEEEEEEE $0\bar{0}1\bar{1}2\bar{2}3\bar{3}$	IIIIIIII RRRRRRRR EEEEEEEE $4\bar{4}5\bar{5}6\bar{6}7\bar{7}$	INHIBITS
9	sp06p1	x01x1xx0xx1x	1x1x1x1x	x0x01x1x	⊤
8	sp05p1	x01xx0x0xx1x	xxxxxxxx	xxxxxxxx	x
7	sp04p1	x01xxxx0xx1x	xxxxxxxx	xxxxxxxx	x
4	sp02p3	x01xxxxxxxx0	xxxxxxxx	xxxxxxxx	1 x
3	sp02p1	x01xxxx0xx1x	xxxxxxxx	xxxxxxxx	x1 x
10	sp07p3	x01x1xxxx0xx	1x1x1x1x	x0x01xx0	xx x
6	sp03p3	x01xxxxxxxx0	xxxxxxxx	xxxxxxxx	xx x 1
5	sp03p1	x01xx0x01x1x	xxxxxxxx	xxxxxxxx	x⊥ x x
1	sp01p1	x01x1xx0xx1x	xxxxxxxx	xxxxxxxx	x• 1 x
2	sp01p3	x01xxxxxxxx0	xxxxxxxx	xxxxxxxx	x11• x
15	sp11p1	x01xx01xx0xx	xxxxxxxx	xxxxxxxx	xxx1 x
11	sp08p1	x01xx01xx0xx	xxxxxxxx	xxxxxxxx	xxx1 x
12	sp08p3	x01x1xxxx0xx	xxxxxxxx	xxxxxxxx	xxxx x
14	sp10p1	x01xx01xx0xx	1x1x1x1x	x0x01x1x	xxx⊥ x
13	sp09p1	x01xx01xx0xx	xxxxxxxx	xxxxxxxx	xxx• x
16	i	x01xxxxxxxx0	1x1x1x1x	x0x01x1x	xxx ⊤x
17	i	x01xxxxxxxx0	x01xx0x0	1x1x1x1x	xxx⊤xx
18	i	xxxxxxxxxxxx	xxxxxxxx	xxxxxxxx	⊥xx11x
19	i	xxxxxxxxxxxx	xxxxxxxx	xxxxxxxx	11 ⊥
30	i	x01xxxxxxxx0	1x1x1x1x	x0x0x01x	⊥x
31	i	x01xxxxxxxx0	1x1x1x1x	x0x01xx0	⊥

Figure 8.7 Logic decode for PUR—the final version

PLA OUTPUT LINES 1 11 111 9874 3065 1251 243	N-SPR 012345	IRE 0123 4567		TASK EQUATION
xxx1 xxxx xxxx xxx	01xxx0	1111 0011		(a→pur)unpk
xxx1 xxxx xxxx xxx	01xxx0	0100 1111		(a→pur)cvb
xxxx xx1x xxxx xxx	01xxx0	1111 0001		(a→pur)mvo
xxxx xx1x xxxx xxx	01xxx0	1111 0010		(a→pur)pack
xxxx xxxx x1xx xxx	01xxx0	???? ????	*	(a→pur)($\overline{\text{mvo+pack}}$)
xxxx xxxx xxxx 1xx	011x0x	xxxx xxxx		b→pur
xxxx x1xx xxxx 1xx	011x0x	1111 0010		(b→pur)pack
xxxx xxxx xx11 xx1	01010x	???? ????	*	(pur→b)$\overline{\text{unpk}}$
xxxx xxxx xxxx x11	01010x	1111 0011		(pur→b)unpk
1x1x 1xxx xxxx xxx	0110x1	1111 0011		(pur→a)unpk
xx1x 1xxx 1xxx xxx	0110x1	???? ????	*	(pur→a)$\overline{\text{unpk}}$
x11x xxx1 xxxx xxx	010011	xxxx xxxx		purh→a;sh→b
x11x 1xxx xxxx xxx	010001	xxxx xxxx		purl→a;sh→b

Figure 8.8 Conventional PLA for PUR—preliminary design

PLA OUTPUT LINES 1 11 111 9874 3065 1251 243	N-SPR 012345	IRE 0123 4567		TASK EQUATION
xxx1 xxxx xxxx xxx	01xxx0	1111 0011		(a→pur)unpk
xxx1 xxxx xxxx xxx	01xxx0	0100 1111		(a→pur)cvb
xxxx xx1x xxxx xxx	01xxx0	1111 0001		(a→pur)mvo
xxxx xx1x xxxx xxx	01xxx0	1111 0010		(a→pur)pack
xxxx xxxx x1xx xxx	01xxx0	xx0x xxxx	*	(a→pur)($\overline{\text{mvo+pack}}$)
xxxx xxxx x1xx xxx	01xxx0	x0xx xxxx	*	(a→pur)($\overline{\text{mvo+pack}}$)
xxxx xxxx xxxx 1xx	011x0x	xxxx xxxx		b→pur
xxxx x1xx xxxx 1xx	011x0x	1111 0010		(b→pur)pack
xxxx xxxx xx11 xx1	01010x	xx0x xxxx	*	(pur→b)$\overline{\text{unpk}}$
xxxx xxxx xx11 xx1	01010x	x0xx xxxx	*	(pur→b)$\overline{\text{unpk}}$
xxxx xxxx xx11 xx1	01010x	1111 0001	*	(pur→b)$\overline{\text{unpk}}$
xxxx xxxx xx11 xx1	01010x	1111 0010	*	(pur→b)$\overline{\text{unpk}}$
xxxx xxxx xxxx x11	01010x	1111 0011		(pur→b)unpk
1x1x 1xxx xxxx xxx	0110x1	1111 0011		(pur→a)unpk
xx1x 1xxx 1xxx xxx	0110x1	xx0x xxxx	*	(pur→a)$\overline{\text{unpk}}$
xx1x 1xxx 1xxx xxx	0110x1	x0xx xxxx	*	(pur→a)$\overline{\text{unpk}}$
xx1x 1xxx 1xxx xxx	0110x1	1111 0001	*	(pur→a)$\overline{\text{unpk}}$
xx1x 1xxx 1xxx xxx	0110x1	1111 0010	*	(pur→a)$\overline{\text{unpk}}$
x11x xxx1 xxxx xxx	010011	xxxx xxxx		purh→a;sh→b
x11x 1xxx xxxx xxx	010001	xxxx xxxx		purl→a;sh→b

Figure 8.9 Conventional PLA for PUR with all necessary rows

In figure 8.10, true and complement lines are represented. No rows or columns have been moved yet. This is a conventional unreduced PLA that will accomplish the function. It has 43 columns and 20 rows for a logic area of 850 units. The PLA in figure 8.10 is the starting point for a PLA folding procedure. First

I fold the AND array, then the OR array. Physical implementation constraints usually require true and complement lines to remain adjacent, but I separate them here to illustrate a procedure for splitting the AND array. (I tried folding the AND array without separating the true and complement lines and could not fold any.)

The idea of folding is to eliminate rows and columns in the PLA by taking advantage of the "don't cares" (represented by *x* in figure 8.10). If, for example, output line 9 in figure 8.9 exits at the top of the PLA and line 8 exits at the bottom, the two lines can be in the same column (the vertical line must be discontinuous between them).

I begin to rearrange the rows and the AND array columns in figure 8.11. I am trying to sort them to leave the largest number of "don't care" terms above or below the last transistor.

Figure 8.12 shows the sample PUR PLA after more row and AND column moves. In figure 8.11, I move the AND columns containing the most transistors to the left and the AND columns

				SSSSSSSSSSS	IIIIIIII	IIIIIIII
PLA OUTPUT LINES				PPPPPPPPPPP	RRRRRRRR	RRRRRRRR
	1	11	111	RRRRRRRRRRR	EEEEEEEE	EEEEEEEE
9874	3065	1251	243	00112233445 5	00112233	44556677
xxx1	xxxx	xxxx	xxx	x01xxxxxxxx0	1x1x1x1x	x0x01x1x
xxx1	xxxx	xxxx	xxx	x01xxxxxxxx0	x01xx0x0	1x1x1x1x
xxxx	xx1x	xxxx	xxx	x01xxxxxxxx0	1x1x1x1x	x0x0x01x
xxxx	xx1x	xxxx	xxx	x01xxxxxxxx0	1x1x1x1x	x0x01xx0
xxxx	xxxx	x1xx	xxx	x01xxxxxxxx0	xxxxx0xx	xxxxxxxx
xxxx	xxxx	x1xx	xxx	x01xxxxxxxx0	xxx0xxxx	xxxxxxxx
xxxx	xxxx	xxxx	1xx	x01x1xxxx0xx	xxxxxxxx	xxxxxxxx
xxxx	x1xx	xxxx	1xx	x01x1xxxx0xx	1x1x1x1x	x0x01xx0
xxxx	xxxx	xx11	xx1	x01xx01xx0xx	xxxxx0xx	xxxxxxxx
xxxx	xxxx	xx11	xx1	x01xx01xx0xx	xxx0xxxx	xxxxxxxx
xxxx	xxxx	xx11	xx1	x01xx01xx0xx	1x1x1x1x	x0x0x01x
xxxx	xxxx	xx11	xx1	x01xx01xx0xx	1x1x1x1x	x0x01xx0
xxxx	xxxx	xxxx	x11	x01xx01xx0xx	1x1x1x1x	x0x01x1x
1x1x	1xxx	xxxx	xxx	x01x1xx0xx1x	1x1x1x1x	x0x01x1x
xx1x	1xxx	1xxx	xxx	x01x1xx0xx1x	xxxxx0xx	xxxxxxxx
xx1x	1xxx	1xxx	xxx	x01x1xx0xx1x	xxx0xxxx	xxxxxxxx
xx1x	1xxx	1xxx	xxx	x01x1xx0xx1x	1x1x1x1x	x0x0x01x
xx1x	1xxx	1xxx	xxx	x01x1xx0xx1x	1x1x1x1x	x0x01xx0
x11x	xxx1	xxxx	xxx	x01xx0x01x1x	xxxxxxxx	xxxxxxxx
x11x	1xxx	xxxx	xxx	x01xx0x0x01x	xxxxxxxx	xxxxxxxx

Figure 8.10 Conventional PLA for PUR with true and complement lines

PLA OUTPUT LINES				SS SSSSSSS IIIIIIII IIII SIIII PP PPPPPPP RRRRRRRR RRRR PRRRR	
1	11	111		RR RRRRRRR EEEEEEEE EEEE REEEE	
9874	3065	1251	243	01 2233455 01234567 2761 40345	
x11x	xxx1	xxxx	xxx	01 x0x0x1x xxxxxxxx xxxx 1xxxx	
xxxx	xxxx	x1xx	xxx	01 xxxxxx0 xxxxxxxx xxx0 xxxxx	
xxxx	xxxx	xx11	xx1	01 x01x0xx xxxxxxxx xxx0 xxxxx	
xx1x	1xxx	1xxx	xxx	01 1xx0x1x xxxxxxxx xxx0 xxxxx	
xxxx	xxxx	x1xx	xxx	01 xxxxxx0 xxxxxxxx 0xxx xxxxx	
xxxx	xxxx	xx11	xx1	01 x01x0xx xxxxxxxx 0xxx xxxxx	
xx1x	1xxx	1xxx	xxx	01 1xx0x1x xxxxxxxx 0xxx xxxxx	
xxxx	xxxx	xxxx	1xx	01 1xxx0xx xxxxxxxx xxxx xxxxx	
x11x	1xxx	xxxx	xxx	01 x0x001x xxxxxxxx xxxx xxxxx	
xxx1	xxxx	xxxx	xxx	01 xxxxxx0 11110011 xxxx xxxxx	
xxxx	xxxx	xxxx	x11	01 x01x0xx 11110011 xxxx xxxxx	
1x1x	1xxx	xxxx	xxx	01 1xx0x1x 11110011 xxxx xxxxx	
xxxx	xxxx	xx11	xx1	01 x01x0xx 111100x1 xx0x xxxxx	
xx1x	1xxx	1xxx	xxx	01 1xx0x1x 111100x1 xx0x xxxxx	
xxxx	xx1x	xxxx	xxx	01 xxxxxx0 111100x1 xx0x xxxxx	
xxxx	x1xx	xxxx	1xx	01 1xxx0xx 1111001x x0xx xxxxx	
xxxx	xxxx	xx11	xx1	01 x01x0xx 1111001x x0xx xxxxx	
xx1x	1xxx	1xxx	xxx	01 1xx0x1x 1111001x x0xx xxxxx	
xxxx	xx1x	xxxx	xxx	01 xxxxxx0 1111001x x0xx xxxxx	
xxx1	xxxx	xxxx	xxx	01 xxxxxx0 x1xxxx11 0xxx x0011	

Figure 8.11 Conventional PLA for PUR after moving rows and columns

PLA OUTPUT LINES				SIISSSSS SS SSIIIIIIIIIIIIII PRRPPPPP PP PPRRRRRRRRRRRRRR	
1	11	111		REERRRRR RR RREEEEEEEEEEEEEE	
9874	3065	1251	243	41724532 01 3502345167620345	
x11x	xxx1	xxxx	xxx	1xx0x10x 01 xxxxxxxxxxxxxxxx	
xxxx	xxxx	xxxx	1xx	xxxx0xx1 01 xxxxxxxxxxxxxxxx	
x11x	1xxx	xxxx	xxx	xxx0010x 01 xxxxxxxxxxxxxxxx	
xxxx	xxxx	xx11	xx1	x0x00xxx 01 1xxxxxxxxxxxxxxx	
xx1x	1xxx	1xxx	xxx	x0xxx101 01 xxxxxxxxxxxxxxxx	
xxxx	xxxx	x1xx	xxx	x0xxxxxx 01 x0xxxxxxxxxxxxxx	
xxxx	x1xx	xxxx	1xx	xx0x0xx1 01 xx1110011xxxxxxx	
xxxx	xxxx	xx11	xx1	xx000xxx 01 1x1110011xxxxxxx	
xx1x	1xxx	1xxx	xxx	xx0xx101 01 xx1110011xxxxxxx	
xxxx	xx1x	xxxx	xxx	xx0xxxxx 01 x01110011xxxxxxx	
xxx1	xxxx	xxxx	xxx	xxxxxxxx 01 x011100111xxxxxx	
xxxx	xxxx	xxxx	x11	xxx00xxx 01 1x11100111xxxxxx	
1x1x	1xxx	xxxx	xxx	xxxxx101 01 xx11100111xxxxxx	
xxxx	xxxx	xx11	xx1	xxx00xxx 01 1x111001x10xxxxx	
xx1x	1xxx	1xxx	xxx	xxxxx101 01 xx111001x10xxxxx	
xxxx	xx1x	xxxx	xxx	xxxxxxxx 01 x0111001x10xxxxx	
xxxx	xxxx	xx11	xx1	xxx00xxx 01 1xxxxxxxxx0xxxx	
xx1x	1xxx	1xxx	xxx	xxxxx101 01 xxxxxxxxxx0xxxx	
xxxx	xxxx	x1xx	xxx	xxxxxxxx 01 x0xxxxxxxx0xxxx	
xxx1	xxxx	xxxx	xxx	xxxxxxxx 01 x0xxxxx111x00011	

Figure 8.12 Conventional PLA for PUR after more row and column moves

with the fewest transistors to the right. In figure 8.12, I attempt to clear transistors from the upper right and lower left corners of the AND array. Once I "diagonalize" the AND array, I cut the vertical lines on the left of the AND array below the last transistor, and I cut the vertical lines on the right side of the AND array above the last transistor. I then fold the PLA by moving as many of the lines from one side of the AND array to the other as the diagonal array will allow.

Figure 8.13 shows the PUR PLA with a folded AND plane. The PLA in figure 8.13 has 34 columns and 20 rows for a logic area of 680 units. This is a 21 percent reduction from the area of the conventional PLA.

Next, I rearrange the OR columns and fold. I fold the AND plane first because it has more transistors. (If I fold the OR plane first,

```
                                SSSSIIIIIIII  SIISSSS
PLA OUTPUT LINES                PPPPRRRRRRRR  PRRPPPP
      1     11 111              RRRREEEEEEEE  REERRRR
9874 3065 1251 243              20150234 5167  4174532

x11x xxx1 xxxx xxx              001xxxxxxxxx  1xxx10x
xxxx xxxx xxxx 1xx              x01xxxxxxxxx  xxx0xx1
x11x 1xxx xxxx xxx              001xxxxxxxxx  •xx010x
xxxx xxxx xx11 xx1              001xxxxxxxxx  10x0xxx
xx1x 1xxx 1xxx xxx              x01xxxxxxxxx  x0xx101
xxxx xxxx x1xx xxx              x010xxxxxxxx  x0xxxxx
xxxx x1xx xxxx 1xx              x01x1110011x  xx00xx1
xxxx xxxx xx11 xx1              001x1110011x  1x00xxx
xx1x 1xxx 1xxx xxx              x01x1110011x  xx0x101
xxxx xx1x xxxx xxx              x0101110011x  xx0xxxx
xxx1 xxxx xxxx xxx              x01011100111  xxxxxxx
xxxx xxxx xxxx x11              001x11100111  1xx0xxx
1x1x 1xxx xxxx xxx              x01x11100111  x•xx101
xxxx xxxx xx11 xx1              001x111001x1  10x0xxx
xx1x 1xxx 1xxx xxx              x01x111001x1  x0xx101
xxxx xx1x xxxx xxx              x010111001x1  x0•xxxx
xxxx xxxx xx11 xx1              001xxxxxxxxx  1x00xxx
xx1x 1xxx 1xxx xxx              x01xxxxxxxxx  xx0x101
xxxx xxxx x1xx xxx              x010xxxxxxxx  xx0••••
xxx1 xxxx xxxx xxx              x010xxxxx111  xx00011

      1     11 111                          SIIIIII
9874 3065 1251 243                          PRRRRRR
PLA OUTPUT LINES                            REEEEEE
                                            3620345
```

Figure 8.13 Conventional PLA for PUR with folded AND plane

it might be more difficult to fold the AND plane.). Figure 8.14 is a conventional PLA with folded AND and OR arrays.

The PLA in figure 8.14 has 27 columns and 20 rows for a logic area of 540 units. This is a 21 percent reduction from the area of the conventional PLA with the folded AND array. It is a 37 percent reduction from the area of the unfolded conventional PLA. It is also 22 percent smaller than the unfolded PLA I implemented with Micro/370.

What would happen if I folded the Micro/370 PUR control PLA from figure 8.7? How would that compare with the folded PLA in figure 8.14? Figure 8.15 is a quickly folded version of the Micro/370 PUR control PLA. I didn't spend much time on it. I didn't even rearrange any rows and columns. I just combined the obvious lines.

```
PLA OUTPUT  SSSSIIIIIIII  SIISSSS
LINES       PPPPRRRRRRRR  PRRPPPP
   11       RRRREEEEEEEE  REERRRR
3 5028617   20150234 5167  4174532

x 1xx1xx1   001xxxxxxxxx  1xxx10x
x x11xxxx   x01x1110011x  xx00xx1
x xx1xxxx   x01xxxxxxxxx  xxx0xx1
1 xxx1xx1   001xxxxxxxxx  -xx010x
1 ----x11   x01xxxxxxxxx  x0xx101
1 xxxxx11   x01x1110011x  xx0x101
x 1xxxxxx   x010xxxxxxxx  x0xxxxx
x x111xxx   001xxxxxxxxx  10x0xxx
x x111xxx   001x1110011x  1-00xxx
x x111xxx   001x111001x1  10x0xxx
1 xxxxx11   x01x111001x1  x0xx101
x xxxx1xx   x0101110011x  xx0xxxx
x xxxx1xx   x010111001x1  x0-xxxx
x x111xxx   001xxxxxxxxx  1x00xxx
1 xxxx-11   x01xxxxxxxxx  xx0x101
1 xxxx1-1   x01x11100111  xxxx101
x xxx1x1-   001x11100111  1xx0xxx
x 1xxxxxx   x010xxxxxxxx  xx0-----
x xxxxxx1   x01011100111  xxxxxxx
x xxxxxx1   x010xxxxx111  xx00011

  111 1                   SIIIIII
 2513944                  PRRRRRR
PLA OUTPUT                REEEEEE
LINES                     3620345
```

Figure 8.14 Conventional PLA for PUR with folded AND and OR planes

LINE	NAME	SSSS PPPP RRRR 0155	S P R 2	S P R 2	S P R 3	S P R 3	S P R 4	S P R 4	INHIBITS
9	sp06p1	011x	1111x1x		00x01x1x				⊤
8	sp05p1	011x	xxxx0xx		0xxxxxxx				x
7	sp04p1	011x	xxxxxxx		0xxxxxxx				x
4	sp02p3	01x0	xxxxxxx		xxxxxxxx		1		x
3	sp02p1	011x	xxxxxxx		0xxxxxxx		x1		x
10	sp07p3	01xx	1111x1x		x0x010x0		xx		x
6	sp03p3	01x0	xxxxxxx		xxxxxxxx		xx		x 1
5	sp03p1	011x	xxxx0xx		0x1xxxxx		x⊥		x x
1	sp01p1	011x	x1xxxxx		0xxxxxxx		x•	1	x
2	sp01p3	01x0	xxxxxxx		xxxxxxxx		x11		• x
15	sp11p1	01xx	xxxx0x1		xxxxx0xx		xxx1		x
11	sp08p1	01xx	xxxx0x1		xxxxx0xx		xxx1		x
12	sp08p3	01xx	x1xxxxx		xxxxx0xx		xxxx		x
14	sp10p1	01xx	1x11011		x0x0101x		xxx⊥		x
13	sp09p1	01xx	xxxx0x1		xxxxx0xx		xxx•		x
16	i	01x0	1•11•1•		•0•01•1x		xxx	⊤	x
17	i	01x0	x01x0x0		1x1x1x1x		xxx⊤		xx
18	i	xxxx	xxxxxxx		xxxxxxxx		⊥xx	11	x
19	i	xxxx	xxxxxxx		xxxxxxxx		11		⊥
30	i	01x0	1x11x1x		x0x0x01x		⊥x		
31	i	01x0	1x11x1x		x0x01xx0		⊥		
			IIIIIII RRRRRRR EEEEEEE 0012233		IIIIIIII RRRRRRRR EEEEEEEE 44556677				

Figure 8.15 Folded Micro/370 PUR control PLA

The PLA in figure 8.15 has 25 columns and 21 rows for a logic area of 525 units. It is 24 percent smaller than the unfolded PLA I implemented with Micro/370. If I eliminate apparently unused lines such as rows 18 and 19, the area becomes 475 units. I could rearrange columns and rows and reduce the PLA even further. Maybe I could save 50 percent of the area. Why didn't I do it in the Micro/370 design?

I have overlooked some area costs in showing you folding PLA. In the Micro/370 implementation, for example, I knew where the input signals came from (physically above and to the right of the space reserved for the PLA) and the exact order and destination of the output signals. If I fold a PLA without regard to the sources and order of input lines and the destination and order of output lines, there could be substantial hidden costs for additional wiring and line crossings.

It may not be possible to separate the true and complement lines in a physical implementation of a PLA, which makes folding less effective. The physical unfolded PLA would have pull-up transistors at the opposite end of the AND column from the input lines. If the PLA input lines are folded, room still must be found for a pull-up transistor for each input line.

The strongest argument for PLA folding is area reduction. The strongest argument against PLA folding is adverse effect of changes. A small change in function may dramatically affect the character of the folded PLA—especially if a program generates the folded PLA. Input and output lines change, affecting wiring. Area may change, which affects placement. Effects of changes in function are amplified if the PLA has been folded.

Special Function Unit Example

The SFU is a catch-all. I use it for all the special things none of the normal pieces of the execution unit can do. It does nibble swaps (inside a byte) for the Pack instruction. It checks legal signs and digits for CVB. It assembles the funny program counter and mask combination for Branch and Link (BAL) and Branch and Link Register (BALR). It is the link between the state sequencer and the execution unit (for access to the instruction registers, the bus status, the program status word—all internal state registers in the state sequencer). It is represented in the execution unit as a 32-bit register that writes to the B bus and reads from the A bus. (I even use it for temporary storage.) It is connected to many parts of the state sequencer by the 32-bit C bus. The flowcharts even assume it is loaded before the first cycle of the current instruction with special values (such as operand length count, masks, and immediate data).

I added SFU to the execution unit on 3 February 1981 as a way to get the mask value from the Test under Mask instruction in IRE into the execution unit. Figure 8.16 shows the list of control lines and transfers specified for the SFU. Figure 8.17 shows the encoding of the nanoword bits for each of the flowchart references to the SFU. The bit patterns were not arbitrarily assigned. I assigned them based on the control lines required to support the transfers. I use a Karnaugh map and try to achieve a placement in the map that results in a single line decoder for each of the transfers. This often is not possible. I usually assign "none" to all zeros in the map (since the first location is arbitrary) and then place the rest of the transfers to achieve minimum decod-

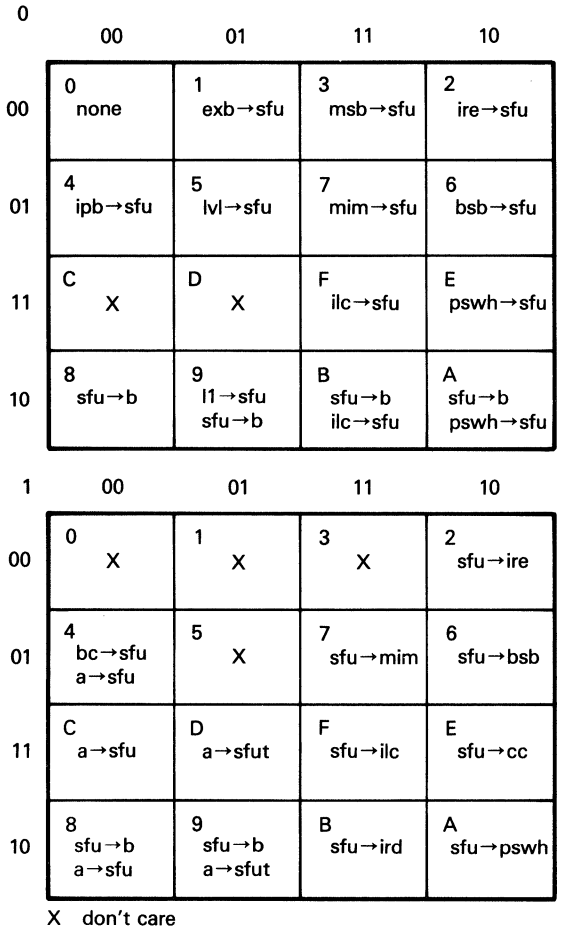
LINE	NAME	TRANSFER	EQUATION
10	sf07p3	ird(0-15)-c(16-31)	(sfu→ird)
19	sf15p3	ire(0-15)-c(16-31)	(sfu→ire)
20	sf15p1	ire(0-15)-c(16-31)	(ire→sfu)
18	sf14p1	ire(8-11)-c(28-31)	(l1→sfu)
1	sf01p3	a(8-31)-sfu(8-31)	(a→sfu)+(a→sfu)(bc→sfu)
17	sf13p3	a(0-7)-sfu(0-7)	(a→sfu).($\overline{bc→sfu}$)
26	sf20p3	sfu(8-31)-c(8-31)	(c→sfu)+a1
27	sf21p3	sfu(0-7)-c(0-7)	(c→sfu)+(bc→sfu)+a1
2	sf02p1	b(0-31)-sfu(0-31)	(sfu→b)
25	sf19p1	sfu(0-31)-c(0-31)	(sfu→c)
16	sf12p3	a(28-31)-sfu(24-27); a(24-27)-sfu(28-31)	(a→sfut)
3	sf03p3	pswh(0-31)-c(0-31)	(sfu→pswh)
7	sf05p3	pswh(18-19)-c(30-31)	(sfu→cc)
4	sf03p1	pswh(0-31)-c(0-31)	(pswh→sfu)
9	sf06p1	ilc(0-1)-c(0-1); pswh(18-23)-c(2-7)	(bc→sfu)
6	sf04p1	ilc(0-1)-c(29-30)	(ilc→sfu)
5	sf04p3	ilc(0-1)-c(29-30)	(sfu→ilc)

Figure 8.16 Execution unit control lines and transfers for SFU

ing. I do an initial placement, move a few things, throw the whole thing away, and start over. After three or four restarts, I usually have something good.

The control lines in figure 8.18 are labeled with a line number and a name beginning with "sf." The circuit designer specified exactly which control lines were needed and gave their order. That's why the line numbers in the PLA in figure 8.18 look random. I designed the PLA and numbered the lines before the circuit designer decided the order of the control lines. The circuit designer reordered the output lines because he designed the SFU and knew which lines he needed to control it and where they must be (physically). (He designed the SFU based on a functional specification taken from the flowcharts.)

Once I had the nanoword bit pattern assignments and the definition of the control lines, I designed the PLA. Line 10 in the PLA (sf07p3 in figures 8.16 and 8.18) controls the SFU→IRD transfer. All I had to do for the control line was find SFU→IRD in the Karnaugh map and decode it. In this case, it took only one line: n-sfu = 11011. Let's look at another. Line 4 (sf03p1 in figures 8.16 and 8.18) controls the PSWH→SFU transfer, which appears twice in the Karnaugh map. I still needed only one decode line: n-sfu = 01x10.



N-SFU	Flowchart Task
00000	none
00001	exb→sfu
00010	ire→sfu
00011	msb→sfu
00100	ipb→sfu
00101	lvl→sfu
00110	bsb→sfu
00111	mim→sfu
01000	sfu→b
01001	l1→sfu;sfu→b
01010	sfu→b;pswh→sfu
01011	sfu→b;ilc→sfu
01100	
01101	
01110	pswh→sfu
01111	ilc→sfu
10000	
10001	
10010	sfu→ire
10011	
10100	bc→sfu;a→sfu
10101	
10110	sfu→bsb
10111	sfu→mim
11000	sfu→b;a→sfu
11001	sfu→b;a→sfut
11010	sfu→pswh
11011	sfu→ird
11100	a→sfu
11101	a→sfut
11110	sfu→cc
11111	sfu→ilc

Figure 8.17 Nanoword bit assignments for SFU control

Look at SFU→B. It is controlled by line 2 in figure 8.16, but it appears six times in the Karnaugh map. I could have covered the six occurrences with two terms: n-sfu = 010xx and n-sfu = x100x, but that would have used two lines, and I only wanted one line in the decoder. I could not have used just x10xx because the terms 11010 and 11011 were occupied. I had to have a line for SFU→PSWH and another for SFU→IRD—the two occupied terms 11010 and 11011, respectively. Since both of those terms had to be there, I decided to let them pull down the line for SFU→B when either term is active. Then I could use a single term for the SFU→B transfer. I call the pull-down lines

from SFU→IRD and SFU→PSWH inhibits. The SFU→B transfer is line 2 in figure 8.18. The SFU→IRD transfer is line 10, and the SFU→PSWH transfer is line 3. You can see the inhibit lines in figure 8.18. One is connected to line 10 and controls the gate of a transistor to ground line 2. Another is connected to line 3 and also controls the gate of a transistor to ground line 2. If either of these lines is high, then line 3 (the SFU→B transfer line) will be pulled low.

The term C→SFU in figure 8.16 is a generic term active for any transfer to SFU that is not (at least partly) from the internal A bus. The term SFU→C is a generic term active for any transfer from SFU that is not to the internal B bus. There actually is a C bus. The transfer SFU→ILC in figure 8.16, for example, should be SFU→C→ILC, but the circuit designers decided to add the "C" bus after the flowcharts were done. The C bus transfers can be implicit in the flowcharts, but they must be explicit in the control logic. I did the flowcharts without the C bus (it does not

SPECIAL FUNCTION UNIT PLA ALMOST AS IMPLEMENTED IN MICRO/370

LINE	NAME	SSSSSSSSSS FFFFFFFFFF UUUUUUUUUU $0\bar{0}1\bar{1}2\bar{2}3\bar{3}4\bar{4}$	$A\bar{A}$ 11	INHIBITS
10	sf07p3	1x1xx01x1x	xx	T
19	sf15p3	1xx0x01xx0	xx	x
20	sf15p1	x0x0x01xx0	xx	x
18	sf14p1	x01xxxx01x	xx	x
1	sf01p3	1xxxxxx0x0	xx	x
17	sf13p3	1xxxxxx0x0	xx	1 x
26	sf20p3	xxxxxxxxxx	xx	x x1
27	sf21p3	xxxxxxxxxx	xx	x xx1
2	sf02p1	xx1xx0xxxx	xx	x11xx
25	sf19p1	1xxxxx1xxx	xx	xx xx
16	sf12p3	1x1xxxx01x	xx	xx xx
3	sf03p3	1x1xx01xx0	xx	x⊥ xx
7	sf05p3	1x1x1x1xx0	xx	x xx
4	sf03p1	x01xxx1xx0	xx	x▪▪xx
9	sf06p1	1xx01xx0x0	xx	+ xx
6	sf04p1	x01xxx1x1x	xx	x xx
5	sf04p3	1x1x1x1x1x	xx	x xx
28	i	xxxxxxxxxx	x0	x11⊥x1
29	i	x0xxxxxx1x	xx	x+x xx
30	i	x0xxxx1xxx	xx	xx+ xx
31	i	xxxxxxxxxx	x0	111 ⊥1
33	i	x0xx1xxxxx	xx	⊥

Figure 8.18 SFU control PLA—similar to the Micro/370 version

appear in the flowchart syntax for SFU control in figure 8.1), but the circuit designers defined control lines for the C bus (lines 25, 26, and 27 in figure 8.16). Control for these lines is still defined by (implicit references in) the Karnaugh map in figure 8.17.

The SFU PLA implemented for Micro/370 has 12 input columns, 6 inhibit columns, 5 inhibit rows, and 17 output lines. The area cost for the PLA is 396 units. How does this stack up against a conventional PLA? This PLA does not depend on the values of IRE (which is running horizontally across the chip), so its funny shape with respect to a conventional PLA may be no advantage (as it was for the PUR PLA). For our implementation of Micro/370, the shape of the SFU PLA in figure 8.18 is advantageous. Since the logic decoders are between the control store and the execution unit, it seems to be advantageous for the inputs to come in at the top of the PLA and for the outputs to leave at the bottom. SFU, however, is located on the right of the execution unit beyond the right side of the control store. The nanoword lines providing the input to the SFU PLA come from the control store and are routed to the left side of the PLA (which is rotated 90 degrees from the figure 8.18 representation). The inhibit lines are below the (sideways) AND array, and the control lines exit above the AND array. (The SFU is above the execution unit.)

The SFU and the SFU PLA are shown in figure 6.1, page 127. The SFU is a large rectangular block just above the right side of the execution unit. The execution unit occupies the lower quarter of the chip. There is a 32-bit bus connecting the SFU to the execution unit. The bus starts in the execution unit, comes out the top of the execution unit, and splits into several smaller segments, which go into the SFU. Just below and to the right of where the vertical 32-bit SFU bus splits is the SFU PLA lying on its side (with the inhibit lines below and the AND array on top).

Conventional PLA SFU Implementation

Figure 8.19 shows a conventional (or standard) PLA for the SFU control. It uses the Karnaugh map placement from the PLA derivation for the Micro/370 PLA. The rows in the AND plane (on the right) of the PLA cover the unique occurrences of terms in the Karnaugh map (see figure 8.17). The N-SFU and A1 columns are the input terms.

The conventional PLA of figure 8.20 has 12 input columns in the AND array, 17 columns in the output array, and 24 rows. The

```
PLA OUTPUT LINES          N-SFU   A1   TASK EQUATION
1121  122  21
0908 1767 2563 7496 5     0 1234  0

xxxx 11xx xxxx xxxx x     1 1x00  x    a→sfu
xxxx xxxx xx1x xxxx x     1 1x01  x    a→sfut
xxxx 1xxx xxxx xxxx x     1 0100  x    (a→sfu)(bc→sfu)
xxxx xxx1 xxxx xx1x x     1 0100  x    bc→sfu
xxxx xx11 xxxx xxxx x     0 0110  x    bsb→sfu
xxxx xx11 xxxx xxxx x     0 0001  x    exb→sfu
xxxx xx11 xxxx xxx1 x     0 1x11  x    ilc→sfu
xxxx xx11 xxxx xxxx x     0 0100  x    ipb→sfu
xx1x xx11 xxxx xxxx x     0 0010  x    ire→sfu
xxx1 xx11 xxxx xxxx x     0 1001  x    l1→sfu
xxxx xx11 xxxx xxxx x     0 0101  x    lvl→sfu
xxxx xx11 xxxx xxxx x     0 0111  x    mim→sfu
xxxx xx11 xxxx xxxx x     0 0011  x    msb→sfu
xxxx xx11 xxxx x1xx x     0 1x10  x    pswh→sfu
xxxx xxxx 1xxx xxxx x     0 10xx  x    sfu→b
xxxx xxxx 1xxx xxxx x     1 100x  x    sfu→b
xxxx xxxx x1xx xxxx x     1 0110  x    sfu→bsb
xxxx xxxx x1xx 1xxx x     1 1110  x    sfu→cc
xxxx xxxx x1xx xxxx 1     1 1111  x    sfu→ilc
1xxx xxxx x1xx xxxx x     1 1011  x    sfu→ird
x1xx xxxx x1xx xxxx x     1 0010  x    sfu→ire
xxxx xxxx x1xx xxxx x     1 0111  x    sfu→mim
xxxx xxxx x1x1 xxxx x     1 1010  x    sfu→pswh
xxxx xx11 xxxx xxxx x     x xxxx  1    a1
```

Figure 8.19 Conventional PLA for SFU with all lines included

area cost is 696 units. That's a lot bigger than the Micro/370 PLA (76 percent), but perhaps the shape helps. Let's go through another folding exercise.

For the PUR PLA folding example, I allowed the true and complement lines to be separated (even though physical implementation probably makes that assumption unrealistic). This led me to count two blank columns in the Micro/370 version (spr0 and $\overline{\text{spr1}}$) that were eliminated in the folded PLA. For this example, I will fold the figure 8.19 representation (rather than the figure 8.20 representation) and double-count the input columns when I figure the area.

Figure 8.21 is the conventional PLA with a folded AND array. Folding did not help much, probably because the decode terms appeared in too many places. Folding works much better with sparse arrays. The PLA with the folded AND plane costs 648 units. It is 64 percent larger than the Micro/370 PLA—not yet an appealing alternative.

PLA OUTPUT LINES 1121 122 21 0908 1767 2563 7496 5	SSSSSSSSS FFFFFFFFF UUUUUUUUU 00$\overline{1}\overline{1}$2$\overline{2}$3$\overline{3}\overline{4}$4	$\overline{A}$ A 11
xxxx 11xx xxxx xxxx x	1x1xxxx0x0	xx
xxxx xxxx xx1x xxxx x	1x1xxxx01x	xx
xxxx 1xxx xxxx xxxx x	1xx01xx0x0	xx
xxxx xxx1 xxxx xx1x x	1xx01xx0x0	xx
xxxx xx11 xxxx xxxx x	x0x01x1xx0	xx
xxxx xx11 xxxx xxxx x	x0x0x0x01x	xx
xxxx xx11 xxxx xxx1 x	x01xxx1x1x	xx
xxxx xx11 xxxx xxxx x	x0x01xx0x0	xx
xx1x xx11 xxxx xxxx x	x0x0x01xx0	xx
xxx1 xx11 xxxx xxxx x	x01xx0x01x	xx
xxxx xx11 xxxx xxxx x	x0x01xx01x	xx
xxxx xx11 xxxx xxxx x	x0x01x1x1x	xx
xxxx xx11 xxxx xxxx x	x0x0x01x1x	xx
xxxx xx11 xxxx x1xx x	x01xxx1xx0	xx
xxxx xxxx 1xxx xxxx x	x01xx0xxxx	xx
xxxx xxxx 1xxx xxxx x	1x1xx0x0xx	xx
xxxx xxxx x1xx xxxx x	1xx01x1xx0	xx
xxxx xxxx x1xx 1xxx x	1x1x1x1xx0	xx
xxxx xxxx x1xx xxxx 1	1x1x1x1x1x	xx
1xxx xxxx x1xx xxxx x	1x1xx01x1x	xx
x1xx xxxx x1xx xxxx x	1xx0x01xx0	xx
xxxx xxxx x1xx xxxx x	1xx01x1x1x	xx
xxxx xxxx x1x1 xxxx x	1x1xx01xx0	xx
xxxx xx11 xxxx xxxx x	xxxxxxxxxx	1x

Figure 8.20 Conventional PLA for SFU with true and complement lines

If I move the rows and columns to try to fold the OR array, it will fold better because there are few transistors in the array. Here is an ad hoc method: Count the transistors in each column. Move the columns with the fewest transistors to the left. Move rows to try to place all the transistors above a diagonal in the array. Move columns again to place some sparse columns on the right. Move rows to place transistors at the bottom of the array. Now you should have something that looks like figure 8.22. Most of the transistors lie close to the diagonal in the OR array. The right half of the OR array can be fitted directly under the left half of the OR array. (It doesn't always turn out so well. When it doesn't, the columns that can't be folded will be in the middle.) In this example, I can fold the OR array almost in half.

Figure 8.23 shows the conventional PLA after folding. It has 10 input columns (I count input columns twice—for true and complement), 9 output columns, and 24 rows. Its area cost is 456 units. It is 15 percent larger than the corresponding Micro/370 PLA.

In the PUR and SFU PLAs, I have shown two design methods and compared the results. In one case, the folded conventional PLA was smaller than the PLA designed using the flowchart method. Why did we still choose the larger PLA? The apparent area used by the AND and OR planes of the PLA is not necessarily representative of the cost of using the PLA. The communication paths for data and control lines in Micro/370 were planned by the logic and circuit designers. We knew where we wanted the PLA, where its inputs were, and where the outputs were going. The PLA was designed to fit in a precise location. The drivers for the inputs were designed to drive the expected loads based on the size of the PLA and the location of the inputs. The output drivers for the PLA also were designed to drive the expected loads the expected distance. We also knew whether the input and output lines would be in metal or poly and left room for them. We did not expect any wires to cross. The wiring was all planned before the PLA was designed.

```
PLA OUTPUT LINES          N-SFU
1121  122  21
0908 1767 2563 7496 5     0123 4

xxxx 11xx xxxx xxxx x     11x0 0
xxxx xxxx xx1x xxxx x     11x0 1
xxxx 1xxx xxxx xxxx x     1010 0
xxxx xxx1 xxxx xx1x x     1010 0
xxxx xx11 xxxx xxxx x     0011 0
xxxx xx11 xxxx xxxx x     0000 1
xxxx xx11 xxxx xxx1 x     01x1 1
xxxx xx11 xxxx xxxx x     0010 0
xx1x xx11 xxxx xxxx x     0001 0
xxx1 xx11 xxxx xxxx x     0100 1
xxxx xx11 xxxx xxxx x     0010 1
xxxx xx11 xxxx xxxx x     0011 1
xxxx xx11 xxxx xxxx x     0001 1
xxxx xx11 xxxx x1xx x     01x1 0
xxxx xxxx 1xxx xxxx x     010x x
xxxx xxxx x1xx xxxx x     1011 0
xxxx xxxx x1xx 1xxx x     1111 0
xxxx xxxx x1xx xxxx 1     1111 1
1xxx xxxx x1xx xxxx x     1101 1
x1xx xxxx x1xx xxxx x     1001 0
xxxx xxxx x1xx xxxx x     1011 1
xxxx xxxx x1x1 xxxx x     1101 0
xxxx xxxx 1xxx xxxx x     1100 •
xxxx xx11 xxxx xxxx x     xxxx 1

1121  122  21
0908 1767 2563 7496 5          A
PLA OUTPUT LINES               1
```

Figure 8.21 Conventional PLA for SFU with folded AND plane

```
PLA OUTPUT LINES      N-SFU
11121   2 22 1
609083745 67571692    0123 4

1xxxxxxxx xxxxxxxx    11x0 1
x1xxxxxx1 xxxxxxxx    1101 1
xx1xxxxx1 xxxxxxxx    1001 0
xxx1xxxxx 11xxxxxx    0001 0
xxxx1xxxx 11xxxxxx    0100 1
xxxxx1xx1 xxxxxxxx    1101 0
xxxxxx1x1 xxxxxxxx    1111 0
xxxxxxx1x 11xxxxxx    01x1 0
xxxxxxxx1 xxxxxxxx    1011 0
xxxxxxxx1 xxxxxxxx    1011 1
xxxxxxxx1 xx1xxxxx    1111 1
xxxxxxxxx xxx11xxx    11x0 0
xxxxxxxxx xxxx1xxx    1010 0
xxxxxxxxx 11xxx1xx    01x1 1
xxxxxxxxx x1xxxx1x    1010 0
xxxxxxxxx xxxxxxx1    010x x
xxxxxxxxx 11xxxxxx    0011 0
xxxxxxxxx 11xxxxxx    0000 1
xxxxxxxxx 11xxxxxx    0010 0
xxxxxxxxx 11xxxxxx    0010 1
xxxxxxxxx 11xxxxxx    0011 1
xxxxxxxxx 11xxxxxx    0001 1
xxxxxxxxx xxxxxxx1    1100 -
xxxxxxxxx 11xxxxxx    xxxx 1

11121   2 22 1
609083745 67571692        A
PLA OUTPUT LINES          1
```

Figure 8.22 Conventional PLA for SFU after moving rows and columns

If I were really a modern designer, I might have fed the specification for the PLA into a design program. The program would automatically design a PLA and then fold it for me. It probably would achieve much better results than the examples above. What does the automatically designed PLA look like? Who knows? The only thing I can say about it for sure is that the inputs can be required anywhere and the outputs can come from anywhere. The general form of this PLA is shown in figure 8.24.

In the examples I used, I (attempted to) split the AND array and OR array only to allow input and output lines access to the top and bottom of the array. There is still another level of compaction possible. If I can rearrange columns in the AND array so some rows have transistors only on the left while others have transistors only on the right, I can eliminate rows by dividing the OR array.

```
PLA OUTPUT LINES    N-SFU      ROW
   2 11121                     NUMBER
   5 60908374       0123 4

   x 1xxxxxxx       11x0 1       1
   1 ·1xxxxxx       1101 1       2
   1 x·1xxxxx       1001 0       3
   x 11x1xxxx       0001 0       4
   x 11xx1xxx       0100 1       5
   1 xxxxx1xx       1101 0       6
   1 xxxxxx1x       1111 0       7
   x 11xxxxx1       01x1 0       8
   1 xx·····        1011 0       9
   1 xxxxxxxx       1011 1      10
   1 xx1xxxxx       1111 1      11
   x xxx11xxx       11x0 0      12
   x xxxx1xxx       1010 0      13
   x 11xxx1xx       01x1 1      14
   x x1xxxx1x       1010 0      15
   x xxxxxxx1       010x x      16
   x 11xxxxxx       0011 0      17
   x 11xxxxxx       0000 1      18
   x 11xxxxxx       0010 0      19
   x 11xxxxxx       0010 1      20
   x 11xxxxxx       0011 1      21
   x 11xxxxxx       0001 1      22
   x xxxxxxx1       1100 ·      23
   x 11xxxxxx       xxxx 1      24

     22 1
     67571692            A
PLA OUTPUT LINES         1
```

Figure 8.23 Conventional PLA for SFU with folded AND and OR planes

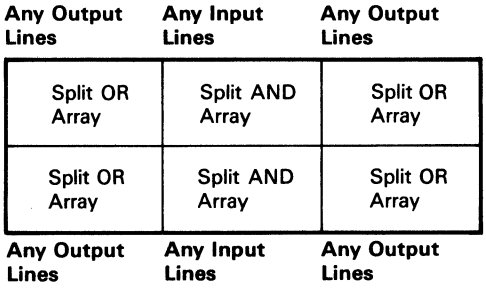


Figure 8.24 General form for a folded PLA

Look at row numbers 16 and 24 in figure 8.23. Line 16 has transistors only on the left columns of the AND array, while line 24 has only a single transistor in the right column of the AND array. Since the two lines do not drive a common column in the OR array, I can move the OR array lines driven by row 24 to the right of the AND array. Lines 16 and 24 can share a common row, with line 16 driving OR array transistors on the left and line 24 driving OR array transistors on the right. Figure 8.25 shows the conventional SFU PLA with folded AND and OR arrays and a split OR array. This new PLA, with an area of 437, has the general form shown in figure 8.24.

As you have seen from the examples and from figure 8.24, the original PLA can be split two ways. I can fold the original AND and OR arrays to share columns, and I can split the output OR array to share rows. Suppose the original PLA has 2X AND array

```
PLA OUTPUT LINES  N-SFU    PLA OUTPUT LINES  ROW
    2 121                      11            NUMBER
    5 908374      0123 4       60

    x xxxxxx      11x0 1       1x             1
    1 xxxxxx      1101 1       •1             2
    1 1xxxxx      1001 0       x•             3
    x x1xxxx      0001 0       11             4
    x xx1xxx      0100 1       11             5
    1 xxx1xx      1101 0       xx             6
    1 xxxx1x      1111 0       xx             7
    x xxxxx1      01x1 0       11             8
    1 ••••••      1011 0       xx             9
    1 xxxxxx      1011 1       xx            10
    1 1xxxxx      1111 1       xx            11
    x x11xxx      11x0 0       xx            12
    x xx1xxx      1010 0       xx            13
    x xxx1xx      01x1 1       11            14
    x xxxx1x      1010 0       x1            15
    x xxxxxx      0011 0       11            17
    x xxxxxx      0000 1       11            18
    x xxxxxx      0010 0       11            19
    x xxxxxx      0010 1       11            20
    x xxxxxx      0011 1       11            21
    x xxxxxx      0001 1       11            22
    x xxxxx1      1100 •       xx            23
    x xxxxx1      010• 1       11            16,24

      1                        22
     571692          A        67
PLA OUTPUT LINES      1  PLA OUTPUT LINES
```

Figure 8.25 Conventional PLA for SFU with split OR array

columns, 2Y OR array columns, and 2Z rows. The area of the unfolded PLA is (2X+2Y)*2Z = 4*(X+Y)*Z. If the AND and OR arrays fold optimally, they will be X and Y columns, respectively. If row sharing also is optimal, the folded PLA will have Z rows. The area of the optimally folded PLA is (X+Y)*Z. Folding, therefore, can reduce the original PLA to a quarter of its original size. In my examples, I didn't do very well in this regard. I didn't get either of the PLAs to even half its original size.

The figure 8.25 PLA is almost as small as the one I was planning to use, so let's put it in the layout. I will have to redesign the drivers from the nanoword. They used to drive lines that ran directly from the output register through the PLA in a straight line of known length. Now some of the input lines have to drive wires around the PLA before they get to the logic line they drive. But because I do not know how the program will fold the PLA, I do not know which of the nanoword bits will be driving the long wires. I will just make all the drivers bigger. The next stop is the PLA output drivers. The old PLA output lines came out of the PLA just where they were needed—at the right location and in the right direction. I knew how big the output drivers had to be. The folded PLA has output lines in all four corners. Any output line can be anywhere. I will have to build PLA output drivers that are big enough to drive a wire around the PLA to the original destination of the output line.

The old PLA fit where I left room for it—just plop it in and wire it up. I can't do that with the new PLA. I can't just wire it up. Because the inputs and the outputs are random, I have introduced a wiring puzzle to solve. Many of the wires will cross, which means they have to run on multiple levels. The old PLA wires did not cross, so they could all run parallel on a single level. Wires that cross and use multiple levels in the design will block area on another level. In the Micro/370 design, we did the wiring first. We knew the wiring was not going to block another level, so we used the other level for power, ground, or another data bus.

Perhaps the biggest drawback to PLA folding is its inability to accommodate changes. The program for folding a PLA might easily generate a new PLA for any changes, but each new PLA could have a different aspect ratio from the last and a completely changed order and positions of input and output lines. These changes can adversely affect chip layout. This would not be as serious if the entire chip was placed and wired automatically as

it would be for a manual design, but the entire chip might have to be generated again to accommodate a change in a single PLA.

When I net out the advantages of automatic generation and folding of PLAs, I get all negatives, which is why I don't use them. Automatic generation and folding is not a good idea if you want to plan the wiring in advance. It's not a good idea if you want to minimize power consumption of the input and output drivers. It's not a good idea if there might be changes. Folded PLAs look smaller, and you might get credit for a smaller PLA if you just count the area inside the PLA. But the real cost of the PLA includes the effect on the input and output drivers, the area to wire the inputs and outputs, plus the area blocked for other circuits.

C H A P T E R

9

VLSI Design Method(ologie)s

We couldn't do much in LSI (large scale integration). Pin and technology constraints limited the complexity of LSI chips to the direct implementation of simple building blocks (such as registers, ALUs, multiplexers, and collections of gates). A few large companies such as Texas Instruments, Fairchild, and Motorola defined the building blocks. They spawned chip families (such as the 54/74 series TTL or the 10K series ECL). As we move from LSI to VLSI (very large-scale integration) we are putting many more transistors on the chip. What are the building blocks now? Chip design is moving into the hands of people who want to put entire subsystems on a chip, not just simple building blocks.

VLSI design is not just moving ahead; I think it is moving in three directions. There are three "cultures" doing chip design for different applications. Unfortunately, the three cultures use the same terms to describe different things. Different methods develop to meet design objectives; that's a good reason for methods to diverge. But because the terms are the same, people begin to view VLSI design as a field in itself. (It is not.) This leads to design methods from one VLSI culture being used to design chips from another VLSI culture, which results in more confusion.

When I first worked on an nMOS chip design (the Motorola MC68000), the design method simply evolved as I went along. I felt too rushed to study design methods at the time, but after I finished, I wanted to go back and study them.

Design methods are founded on design objective, computer history, technology, and designer's preferences. My interest is designing single-chip processors. For this, I want to know in detail how you convert formal, readable English text (such as IBM's *Principles of Operation*) to the formal, unreadable notation of a typical logic design. I have not found adequate descriptions of how this is done. Here, then, are my observations on this process. I define three ways of designing chips and call them methods A, B, and C.

Method A: The Machine Partition Method

Method A builds chips using off-the-shelf items called macros, which we know how to design well (examples are ALUs, registers, and PLAs). A computer program can position, or place, these pieces on a chip and connect them with wires. Linear programming is used to help place the pieces since the best placement achieves several objectives. The placement job is difficult because there are *n* factorial ways to place *n* pieces. The wiring can be so complicated it is hard to find even a single complete wiring solution. Wires often consume more than 50 percent of the chip's area.

Method A develops in companies designing large processors. Lots of people work on the design. The processor is partitioned into functional blocks, with each block assigned to a different group. Design levels, too, are partitioned, and each level is assigned to a different group. Architecture, specification, logic design, circuit design, and layout may all be separate parts of the organization.

Method A design begins with a block diagram of the controller and execution unit. The block diagram is then partitioned among the logic designers. Circuit designers provide a catalog of macros for use by the logic designers. This catalog is much like a TTL (transistor-transistor logic) parts catalog. It is good for the logic designers because their job does not change with technology developments. (The catalog may evolve with technology, but

the logic design procedures need not.) Automatic placement and wiring programs take the logic design and produce the chip layout. Each group works independently; formal documents link groups. There is a distinct separation between logical design and physical design.

IBM commonly uses method A. I believe method A is necessary for large gate array computers. Such computers require thousands of chips, so the design must be partitioned early. IBM uses method A in conjunction with bipolar gate array chips (masterslices). Typically, gate array chips contain arrays of NAND or NOR gates (hence their name), which can be wired together (personalized) in a few manufacturing steps. If your logic is expressed in NANDs or NORs, your group can essentially design hardware without circuit designers. Since gate array chips let manufacturers create many part numbers from the same basic chip, they are cheaper to produce.

IBM has developed many design automation programs for its multi-chip gate array designs. I think this has encouraged the use of method A for IBM's single-chip designs. The single-chip System/370 is an example (Davis, 1980). I think using method A for a single-chip design adds to implementation difficulties. The method divorces the processor's physical appearance from what the processor does. (The processor's block diagram and the chip do not look the same.) This happens because the placement and layout of macros is independent of how or where they are to be used. Information about inherent communication paths in the processor is lost in the data base of independent macros. Loss of information about inherent structure of communication paths is likely to result in a less efficient physical implementation.

The independent macros are optimized individually for size and shape. Shrinking macro layouts and folding PLAs add to the wiring congestion by making the location of input and output lines more random. The computer constructs a chip plan using information about the size, shape, and interconnection of macros. Macros could be better optimized if location in the communication path was considered. The computer cannot reconstruct the communication path as efficiently using independently optimized macros because they may not fit together well.

Transistors at the output of a macro must be large enough to drive the next logic stage no matter how far away the placement

program places it. This means macros all have large output drivers, which are bigger, use more power, and have higher capacitance. They slow down the logic that drives them.

The flow of data and control signals is determined independently of the logic design, which results in a vastly increased area for wiring. In a complicated design, this increase is enough to bump large amounts of capability off-chip. The separation of logic design from circuit design and layout is necessary in big multi-chip designs (so you can partition the design and mobilize your cast of engineers), but I think it is a mistake for single-chip designs.

All companies that sell commercial single-chip processors as an end product use a design method that preserves the global data and control signal flow on the chip. The instruction decoder is near the pins where instructions arrive. The controller logic is driven directly by the instruction decoder. The execution unit (including registers) is adjacent to and aligned with the controller's output lines. Functions are not separated or located randomly and then wired. Pin assignments are based on the need to preserve the global information flow. All this reduces wiring overhead (with its decreased speed and increased power dissipation.)

Method A chips do not have the preservation of global information flow as their major design goal. As a giant processor's block diagram is partitioned into chips, the global data and control signal flow are preserved all the way down to the chip level—frame, gate, board, and card. No one would think of allowing random placement of chip modules in a frame. It would be a disaster. Signal lines running in and out of the chip attempt to preserve the global information flow for the giant processor. Only within the chip is the wiring arbitrary. Companies using method A to design single-chip processors have forgotten to make the preservation of global information flow the major goal. They throw away the information about global communication and later let the computer try to reconstruct it. Doing so leads to chips with lots of wiring, less function, and high power dissipation.

I believe that communication among groups is a serious problem in large design efforts because it is limited to formal documents. It's not that groups don't talk informally. It's that under pressure, the transfer of formal documents causes problems. The logic design manager is measured on how the logic designers do from

Method Attributes	Method A Machine Partition
"Books"	Fixed
Companies	IBM Amdahl CDC DEC
Floor plan	No
Logic synthesis	Yes
Macros	Fixed
PLA folding	Yes
Silicon compiler	Yes
Standardization	High
Technology dependent	Low
Transistors	Fixed sizes

Result Attributes	Method A Machine Partition
Communication loop	No
Density	Low
Design cost	Medium
Function	Low
Placement	Difficult
Power	High
Speed	Medium
Wiring	Difficult

Figure 9.1 Machine partition method applied to single-chip microprocessor design

formal-document-in to formal-document-out. This leads to verification at each level; the next manager will not accept an unverified design document. The result is overkill on testing and verification. Everyone wants to be able to point to someone else if something goes wrong.

The main objectives of method A design are maximum standardization and logic independence. These objectives are necessary for giant processor design. Some companies, such as IBM, have attempted to use method A to design single-chip microprocessors. My view of the machine partition design method as applied to the design of a single-chip microprocessor appears in figure 9.1.

Method B: The Commercial Microprocessor Method

Method B is used by companies selling microprocessors as components in commercial competition. The livelihood of companies such as Motorola, National, Signetics, Intel, and Zilog depends on whether their VLSI logic components are commercially competitive. Their designs must meet rigid requirements for maximum performance and function with minimum area and power.

One variation of method B builds chips by trying to "cover" the Boolean equations describing the processor with logically equivalent equations containing the minimum number of Boolean terms. Each of these terms prescribes custom circuits that minimize the number of transistors. This variation of method B came from the days when chips were expensive and simple and designs using these chips were constrained by chip cost. The best minimization of gates required the fewest chips and therefore the cheapest design. Chip designers adopted these procedures even as integrated circuits became dense and cheap. The Zilog Z8000, the Motorola MC6809, and the Intel 8080 were implemented using this variation of method B. (I call these designs spaghetti chips after the physical resemblance.) Devotees of this method claim it produces the fastest chip in the smallest area. Even the spaghetti chips preserve the global flow of data and control.

The "spaghetti" method worked well for simple chips such as the early hand-calculator chips. It also was appropriate for early multiple-chip designs because the cost of logic was high. I suspect designers using the "spaghetti" method of having experience in one of these areas.

Correcting or modifying a "spaghetti" method design is difficult. (It is much like trying to correct an error in an intricate wood carving.) As you put more and more transistors on a chip, it becomes more and more difficult to find a minimum set of Boolean terms.

I do not think the "spaghetti" method is good for present-day single- or multiple-chip designs. Experienced designers know that much of their time is spent correcting errors or making design changes. The difficulty of making changes in "spaghetti" designs makes them less acceptable as technology advances.

Since the design is based on a minimum cover of the Boolean equations, timing signals appear among the implemented terms at places dictated by the minimization procedure. This raises the chances for timing errors and race conditions if clock terms are skewed by different logic delays.

Perhaps the most serious problem with the "spaghetti" method is the amount of work to be done by hand. Each minimum term can have a unique physical realization as a complex circuit of a

hundred or more transistors. This method is the least amenable to help from computers, which can be used to copy, extend, and modify similar circuit designs to create larger, higher-level logic functions (such as PLAs, ROMs, and register files).

The flowchart method is a variation of method B. It is based on the belief that implementation structures (such as the execution unit, controller, and microword format) are implicit in the English description. A designer must find them, not impose them. The premise is that the quality of the design is adversely affected when implementation structures are selected too early. By quality I mean the size, speed, cost, power, and ease of implementation. The flowchart method does not assume that the English description implies one best implementation.

When all I could fit on one chip was an ALU, I had a good handle on what the chip was trying to do, so I could jump right into the intricacies of ALU design. Now I can fit so much function on a VLSI chip that even its English description defies comprehension. The extent of my understanding is that I am designing a processor that fetches and executes instructions residing in memory. What do I do with that and a three- or four-hundred-page English description? I can't expect to work out the states simply by staring at the specification for a while.

Popular Belief

The form of physical design—microprogrammed, random logic (hardwired), or PLA—implies how closely the design matches the problem. So if I hear that an implementation is hardwired, I immediately believe that it is faster than a microprogrammed implementation. The minute I hear "microword" or see that ROM on the chip photograph, I know in my gut this processor is slower than it would be in random logic. But the microcoded structure makes it much easier to debug and much easier to change, so it is worth it. PLAs are a compromise between microprogrammed and hardwired implementations. PLAs give a structure that is easy to debug and change; and PLAs, like random logic, are custom-tailored to the execution unit. Figure 9.2 shows why, I think, we've come to believe this garbage. Anyone can see random logic has to be faster.

Fundamentally, you always have these four things: instruction decoder, controller, execution unit decoder, and execution unit (figure 9.3). The degree to which these four things can be

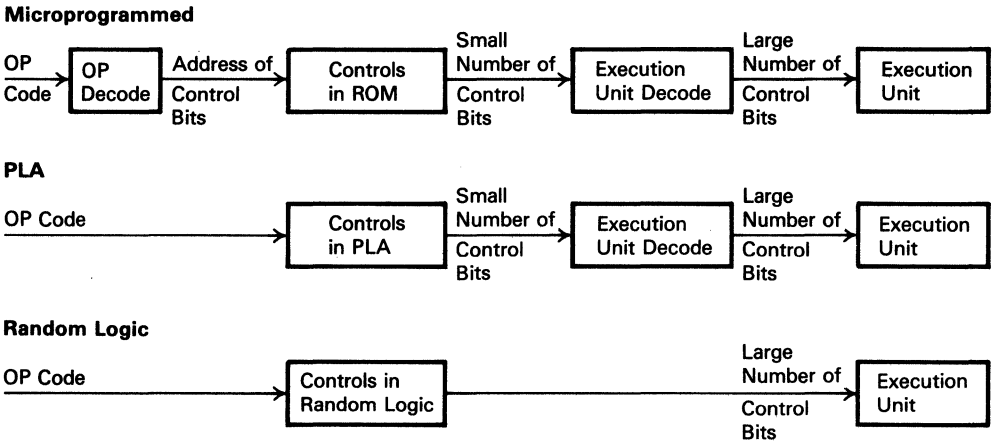


Figure 9.2 Forms for microprocessor implementation

merged in an implementation is now thought to be due to how they are implemented—that is, in microcode, random logic, or PLAs. This just isn't the case.

Whether the controller input is the input to a ROM, a PLA, or to random logic does not tell you whether you can or want to smash everything together or how the controller was created. (See figure 9.4.)

Random logic and PLA implementations, I believe, are designed in parallel with the execution unit—that is, they are custom-fit to the execution unit. But microcoded implementations obviously come after the execution unit. In the System/370, the execution unit is a given, and it is usually (micro)programmed to make it do what we want. We start with given operations, and the problem is to sequence those given operations.

The Flowchart Method

With the flowchart method, the idea is to specify controls and an execution unit at once in a highly organized, visual form. These controls can be implemented as bits in a ROM, as a PLA, or as random logic. The flowchart notation can be mapped into ROM bits, PLA terms, or Boolean equations.

In the flowchart method, the designer selects an initial execution unit and flowcharts the instruction set. The execution unit sees

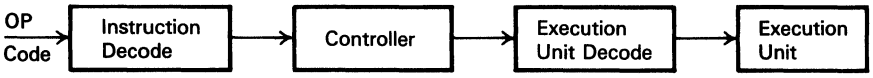


Figure 9.3 Four fundamental processor functions

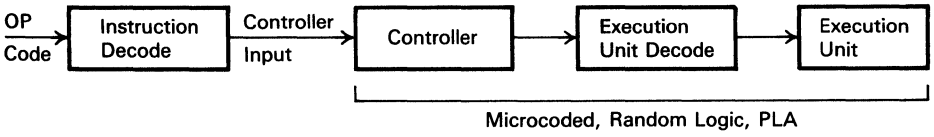


Figure 9.4 Required functions do not depend on implementation method

many revisions as flowcharting progresses. These flowcharts are not like software flowcharts. Each box in a hardware flowchart is a processor state and consists of one or more source-to-bus-to-destination transfers among execution unit elements. These flowcharts are formal enough to describe the processor unambiguously to circuit designers. Further, you can implement the processor using microcode, combinational logic, or PLAs by a mechanical translation of the same flowcharts. Six pages of flowcharts completely described the controller for the MC68000 microprocessor.

The data and control signal flow for the processor are taken into account with the development of the formal description of what the processor does. Decisions about how the design should be partitioned for implementation can be made on the basis of what actually will be happening in the processor. This is because the flowcharts let you see what the processor will be doing (before you select any structures).

Structures based on what you actually see happening lead to a higher quality implementation. The structures (in both execution unit and controller) evolve considerably during the design. For example, the MC68000 register file grew from eight 16-bit data registers to eight 32-bit registers, and other registers were moved or added in the execution unit. Similar changes took place in the MC68000 controller without seriously affecting the design schedule or design quality.

Why is this method good for VLSI? The methods used to design simple building blocks are just not adequate for chips with complicated functions. What I need is a method that works for complicated chips, one that lets me find the right structures systematically and does not require seeing or imposing them at the outset. The flowchart method does this.

The flowchart method is a good way to make implementation decisions because you see design weaknesses when you still have plenty of time to do something about them. It does this by giving you an organized visual representation of the processor. If you have never written anything down in an organized way, you may not appreciate the value of doing so. The act of putting something into organized form gives insight into how things are related.

The flowchart method works because it is organized and visual. Flowcharts let you use your natural ability at pattern recognition to see the relationships within and among instructions. You see the contents of processor states. You view the design on many levels from one representation. You base important design decisions on the best way to implement patterns you recognize as similar. These patterns suggest structures. A structure (such as a processor block diagram) imposed before the flowcharts are available interferes with natural patterns in the processor's operation. With the flowchart method, you can:

1. Accurately compute the performance of the chip
2. Accurately compute external bus utilization
3. Accurately compute the size of the control store and other structures
4. Find common logic sequences (to reduce the control store size)
5. Find common states (to reduce the control store size)

And you do these things before you commit to a controller design. The flowchart method characteristically tolerates major design changes. Rather than trying to impose structure, the flowchart method lets you find the correct structure for the chip. I think the flowchart method is the one to use for single-chip designs. My view of the commercial microprocessor method applied to the design of a single-chip microprocessor is shown in figure 9.5.

Combinational designs currently are viewed as bad compared to the "regular" PLA and ROM (microcoded) designs. Regular de-

	Method B
Method Attributes	**Commercial Microprocessor**
"Books"	None
Companies	Intel Motorola National Signetics
Floor plan	Yes
Logic synthesis	No
Macros	Custom
PLA folding	No
Silicon compiler	No
Standardization	None
Technology dependent	Yes
Transistors	Any size

	Method B
Result Attributes	**Commercial Microprocessor**
Communication loop	Yes
Density	High
Design cost	High
Function	High
Placement	Simple
Power	Low/Medium
Speed	Fast
Wiring	Simple

Figure 9.5 Commercial microprocessor method applied to single-chip microprocessor design

signs fit better with automated design techniques, but relative size and speed advantages are difficult to prove. Combinational designs evolved from design of simpler processors and are appropriate for these. Microcoded designs are appropriate for processors using ROM or RAM as a controller building block or for processors that need microprogrammable controllers. PLA designs are good for VLSI because they are compact and chip area is at a premium.

Method C: The Logic Replacement Method

Method C favors pieces such as ALUs, registers, and PLAs constructed so they are efficient for the technology. Method C tries to find and use pieces that are easy to implement, whereas method A uses pieces that are logically elemental (technology independent). Method C pieces are large, variable geometry units altered so they fit together well for the flow of data. A floor plan is chosen for the data paths. The aspect ratios of components (such as registers, shifter, and ALU) are altered to make data paths uniform (straight runs) and to accommodate buses and signal lines. Changing aspect ratios to make data paths uniform introduces spaces in many of the components. Control lines get longer (making the chip slower). The chip may be too

sparse to be competitive. Choosing a controller and execution unit (via a floor plan) is bad because the controller and execution unit do not exactly suit the requirements of the specification. A floor plan also imposes a control structure and execution unit layout on the chip. These are things that should evolve; they should be discovered as the design progresses, not be imposed at the outset.

The SCHEME chips (from MIT) and OM1 and OM2 (from Cal Tech) are method C chips. So are the various RISC, SOAR, and SPUR chips from the University of California–Berkeley and the MIPS chip from Stanford. This is the method described by Mead and Conway (1980) in their book *Introduction to VLSI Systems.* The physical efficiency of having the control signals run orthogonal to the data paths in a two-dimensional technology (such as MOS or bipolar) is recognized and used. Other techniques (such as current steering and transistor stacking) are recommended because of their advantages for nMOS technology. Method C advocates building subsystems on a chip using structures suited to the specific technology. It lends itself to computer help because the circuits frequently are copies or moderate variations of other circuits.

The emphasis of method C is minimum design effort. Speed, power, and density are not important. Method C is best for integrating existing designs to increase reliability or reduce manufacturing cost. Many of the ideas proposed by method C advocates are applicable to commercially competitive designs, but many of the reported benefits are lost when speed, power, and density become important. My view of the logic replacement method as applied to the design of a single-chip microprocessor is shown in figure 9.6.

Summary of Methods

As I said before, I think there are three "cultures" of VLSI design based on methods that produce chips for different objectives. Method A, the machine partition method, is based on the fundamental objectives of maximum standardization and logic-independent design. It is appropriate for designing chips for a giant partitioned project. Method B, the commercial microprocessor method, is based on the fundamental objective of maximum function per unit area. It is appropriate for designing commercially competitive, high-volume logic chips. Method C,

	Method C
Method Attributes	**Logic Replacement**
"Books"	Variable
Companies	Universities VTI ZyMOS Citel
Floor plan	Yes
Logic synthesis	Yes
Macros	Variable
PLA folding	No
Silicon compiler	Yes
Standardization	Some
Technology dependent	Medium
Transistors	Any size

	Method C
Result Attributes	**Logic Replacement**
Communication loop	Yes
Density	Low
Design cost	Low
Function	Low
Placement	Simple
Power	Low/Medium
Speed	Slow
Wiring	Simple

Figure 9.6 Logic replacement method applied to single-chip microprocessor design

the logic replacement method, is based on the fundamental objective of minimum design cost. It is appropriate for designing low-volume logic replacement chips. Companies specializing in this design style call these chips application-specific integrated circuits (ASICs). Each of these methods has its own area of appropriate application. Figure 9.7 is my summary of the method attributes of the method and the result of applying the three methods to single-chip microprocessor design.

Contrasting Folklore with Reality

All the methods have been used to design single-chip microprocessors. All the methods are called VLSI design. People working in "VLSI design" often are not aware that the other "cultures" are so different. The terms used by experts in each of these cultures are familiar to the experts in other cultures. It's like having three different languages that all use the same set of words. You can recognize the words, but the meaning is actually different for different languages. This creates confusion and controversy and costs companies a lot of money and wasted effort.

VLSI design is a hot topic. Companies spend—and make—a lot of money on it. Students are interested in it. Professors can get government grants to work on it. Junior faculty members can

	Design Method		
Method Attributes	**Machine Partition**	**Commercial Microprocessor**	**Logic Replacement**
"Books"	Fixed	None	Variable
Companies	IBM Amdahl CDC DEC	Intel Motorola National Signetics	Universities VTI ZyMOS Citel
Floor plan	No	Yes	Yes
Logic synthesis	Yes	No	Yes
Macros	Fixed	Custom	Variable
PLA folding	Yes	No	No
Silicon compiler	Yes	No	Yes
Standardization	High	None	Some
Technology dependent	Low	Yes	Medium
Transistors	Fixed sizes	Any size	Any size

	Design Method		
Result Attributes	**Machine Partition**	**Commercial Microprocessor**	**Logic Replacement**
Communication loop	No	Yes	Yes
Density	Low	High	Low
Design cost	Medium	High	Low
Function	Low	High	Low
Placement	Difficult	Simple	Simple
Power	High	Low/Medium	Low/Medium
Speed	Medium	Fast	Slow
Wiring	Difficult	Simple	Simple

Figure 9.7 Summary of attributes and results for chip design methods

publish articles on it, which leads to promotions. There is frenetic activity in VLSI design, as many people try to solve the same general problems. We have the Wiring Problem, the PLA Folding Problem, the Pin Limitation Problem, the Interconnect Problem, the Logic Synthesis Problem, the Logic Verification Problem, and so on.

Each of these problems is attacked and results reported as if it related to VLSI design in general. But a problem usually belongs to one design culture. The Wiring Problem and the Placement Problem belong to the machine partition culture (method A).

Even a problem belonging to more than one culture probably has different solutions for each culture. The Power Dissipation Problem might be solved in the machine partition culture by improving cooling and in the commercial microprocessor culture by reducing function or speed.

Reported solutions rarely distinguish among the cultures. I work in the commercial microprocessor culture. Solutions purporting to solve my problems in VLSI design (some problems I did not even think I had) cause me trouble. Wild benefit claims interfere with my work. I wish I could ignore them, but I work for a company. Company management and their staffs attempt to find out what is going on in VLSI design. They read the wild claims. They want to know why I am spending all their time and money on "old" design methods and tools when Professor Figby at Big U has an automatic chip-generation program as good as the Motorola MC68000 design team's that runs for four dollars on a pocket calculator. I stop work to talk to the staffie. I study Figby's miracle method and find it applies only to method C designs (and does not even work on *them*). I then respond to the executives and the staffies. Result: I'm not working on my chip.

One of the exaggerated claims made over the past few years announced that the time and effort involved in the design and layout of commercial microprocessors like the Zilog Z8000 or the Motorola MC68000 was going to shrink from some twenty man-years to a few man-weeks. What would cause this to happen? A new automated design system, of course. There you have it: VLSI design is easy and automatic. Fast too. Not only that, this new system is right in my area, commercial microprocessors. You think management won't notice this? I figure it costs more than twenty man-years to design a commercial microprocessor. At about $100,000 per man-year, that comes to more than $2 million (I think it costs closer to $5 million). But the managers look at a few man-weeks at, say, $20,000 (total) and want to know why this project is costing $5 million.

If the new system lived up to the predictions made for it, the whole United States would be hip deep in 68000-class microprocessors by now. Nobody seems to be having trouble wading through all that high-performance silicon. I haven't cut my shin on any stray chips. In fact, I haven't seen a single chip come out

of that miracle design system. I didn't think I would. But I couldn't prove it several years ago when the announcement appeared and management wanted an answer. The managers and their staffs didn't know that the claim was unreasonable.

Other claims made for design automation efforts are that they increase a designer's efficiency and lower costs—even that they can turn any system designer into a chip designer. The claim that more nearly reflects reality is that the design changes to fit the tool. That's how I feel about it too. I think that many design automation people don't understand the methods they are automating. They try to guess what will help or how to apply the latest technology. No time is given to studying the methods that designers prefer to use. I know only one design automation fanatic (Dick Hadsell) who studied a designer's method before beginning to implement tools.

Still another claim, for an automated standard floor-plan system, had this system approaching or even exceeding human design proficiency. This may be so for methods A and C, but it's not so for method B. It isn't even close. No commercially competitive microprocessor has ever been designed using automatic layout and a standard floor plan. But it looks cheaper and faster, and somebody is saying that it's better than design done by humans. So the managers and staff members see this claim in print, and they worry about the dollars going into the fire. They want to know why, so I have to stop working on the design to tell them.

Here I will have to accede, though. It might be possible to produce a commercially acceptable part, but the floor-plan system could never come close to the proficiency of someone good at commercial microprocessor design. (You can get a commercially acceptable part using an inefficient method if it is the only available part or copies a design with an established market. Digital could do this with the VAX architecture. IBM could do it with System/370.)

Finally, there is a claim that with a silicon compiler, a designer can simply describe the IC elements to a computer, and the silicon compiler will automatically do the layout. This method wouldn't be good enough for a commercial microprocessor, but it sounds okay for logic replacement or even for the machine partition culture.

Conclusion

What's the point of all this? There are staffies and managers around who read about these miracle solutions and want to know why your group hasn't gotten the word. If you don't know all this is going on, it speaks well of how you work. Maybe you have a good manager, or maybe your group has a political deflector. (The political deflector is hired by your group, but he doesn't work on your project. He works full-time to make the project down the hall look like the "hottest" advanced development project in the company. That helps steer the bozos, staffies, and managers away from your project and toward someone else.)

What do you do if you are caught up in this mess? I have seen four solutions.

1. You can ignore the interference. This solution is used by Seymour Cray. If you aren't Mr. Cray, it is difficult to apply this solution. (Unless you work for yourself—in which case I think it is the best solution.)
2. You can move about every three years. If you change companies, the staffies and managers will be hounding someone else until they find out you are there. It takes about three years for the burden to become intolerable. This is the most common solution.
3. You can try to stay unknown, which is difficult if you want interesting and challenging work and are very good at it.
4. You can get a good manager. This solution is almost impossible. That's where I am as I write this, but my manager might be wearing out. He is taking the entire burden of trying to protect the whole group so they can work. He has to be good, and he doesn't do any technical work.

References

C. Davis et al., "Gate Array Embodies System/370 Processor," *Electronics,* 9 October 1980, pp. 140–43.

Carver Mead and Lynn Conway, *Introduction to VLSI Systems* (Reading, Mass.: Addison-Wesley, 1980).

E P I L O G U E

A Final Word

T. A. Welch

As a former supervising professor of Nick Tredennick, I feel the need to exercise a latent right to criticize, explain, and apologize for his technical writings, particularly in this case. He has written an interesting book but has not explained well what lessons the reader should draw from this experience. Let's try to figure out what they are.

This diary of a design process documents the use of a particular computer organization (microprogrammed, bus-centered) for a particular computer architecture (IBM System/370). How much of this process is inherent to designing computers in general? How much of it is dictated by the particular design objectives of this VLSI chip? How much of it is due to Tredennick's idiosyncrasies? The design given here is quite a good one for a VLSI 370 and certainly should be studied by all people expecting to design VLSI 370s. But that is a limited job market. Most computer designers will be working on controllers geared to high-performance or peculiar data operations, since these are the demands that justify special processor designs.

What Did He Say?

The Tredennick design nicely illustrates the following three elements of processor design.

1. **Microcode compaction.** If you are building a microprogrammed machine (a reasonably common event if you are not into RISC designs or super high-performance machines) and if you are working under the tight space constraints of a VLSI chip, then microcode compaction is important. This book gives a good discussion of alternatives there. The design style for the Micro/370 results in very high density and is worth noting.
2. **Chip layout.** The Micro/370 design is a good example of the "floor-plan" style of chip design, namely method B of chapter 9. As Tredennick says, interconnects take up more space than logic elements on VLSI chips, so designing the interconnect structure is as important as minimizing the logic. The most space-efficient designs are achieved by altering the logic to fit the geometry. The level of designer effort needed to do that is justified for high-volume chips where manufacturing yield plays a role in the economic success of the design.
3. **Microcoding process.** Reducing a processor's instruction set to a sequence of register-transfer events is a basic process in machine design. It looks like programming in the sense that functions are being implemented by a sequence of operations involving conditionals and iterations. The differences are significant, however. First, the microprogram is controlling tightly synchronized concurrent activities. At times, this work is closer to choreography than programming. Present popular programming languages are ill suited to this kind of computing. Second, the microprogrammer has an opportunity to adjust the microinstructions. Tredennick does that by going back and modifying the execution unit. The dual challenge of writing an efficient program while refining the underlying instruction set is an interesting one. This book illustrates that process well.

What Did He Miss?

Tredennick had an easy job because the instruction architecture he implemented is well defined and stable. No programmers were coming to him asking for changes to speed up critical routines. No simulations of the instruction set were needed to

make sure the instruction definitions actually worked right. This instruction set has been implemented many times before, so the general techniques for building its trickier features are part of the folklore of the art. People who build special-purpose processors do not have the advantage of having an instruction set that does not change in the course of the implementation.

The primary characteristic of the IBM System/370 instruction set is that it is complicated. This pushes the implementation into the direction of using microcode as the most efficient means to store the necessary logic. Simple instruction sets, as are typically found in minicomputers, give more freedom of design choices.

The larger factor that simplified this design effort is the lack of pressure to achieve very high performance. This design focuses on minimized chip area rather than on achieving highest speed, so some design problems that are important in most design efforts are not seen in this one. These concern timing issues, which occur when squeezing a few nanoseconds out of the ADD instruction becomes important. The book is notable because it contains not a single timing diagram, which would show the sequencing of processor events in the course of a clock cycle on a nanosecond-by-nanosecond basis. Likewise, the book does not show the execution of two instructions in a row, preferring always to show the events of only one instruction at a time. This is because Micro/370 instruction executions are independent of each other, as opposed to the overlapping of instruction executions common in faster implementations.

When building a high-performance implementation, a designer typically works out the exact timing of two critical paths in addition to individual instruction execution timings:

1. Execution of a conditional branch instruction, where the branch is taken and the data condition being tested was calculated in the immediately prior instruction
2. Address index calculation, where the index register value was calculated in the prior instruction

The fact that these timings pose no problems in the Micro/370 and are not discussed in this book is a sign of the relatively leisurely pace of the Micro/370 executions.

The net result of these biases is that several interesting types of processor design are not discussed in the book.

Alternative Processor Styles

Let's compare the Micro/370 to other common design styles by looking at basic components of any such design.

1. **Control logic.** The Micro/370 uses a two-level control store, which is an unusual processor structure. Most designs substitute custom logic for one or more level. The two-level design achieves the densest packing of control information, at the sacrifice of slowness in the translation of processor state into actual execution unit events. The slowness of translation causes delays when the next processor action is determined by some data criterion. After the data is tested, the execution unit essentially must be idle until the data condition is processed by the control unit into the next set of microcommands.

2. **Register interconnect.** The Micro/370 uses a bus for most data transfers between registers. This can cause a slower design for two reasons: The bus is a critical resource, and some transfers have to be delayed until conflicting demands for the bus are satisfied; the bus is electrically slower than direct register-to-register transfer paths because of the large number of loads (and large capacitance) it must drive. The bus saves space and permits easier modification of instruction execution logic, but these gains are at the sacrifice of speed.

3. **Cache memory.** Many processors now use a form of local memory known as cache to store recently referenced data for possible reuse. The well-coordinated integration of a cache into processor sequencing raises multiple design problems. In Micro/370, there was no room on the chip for a cache, so this interesting topic did not find its way into the book.

4. **Pipelined execution.** High-performance processors generally take advantage of the fact that instruction executions have the following four phases, which utilize different resources: instruction decode, using control logic; operand address calculation, using index registers and an adder; operand fetch from memory, which does not use any part of the execution unit; operation execution, which uses the arithmetic unit. By implementing these phases in separate units, four different instructions might be in different stages of execution at the same time. A processor built this way can go as much as 150 percent faster than a nonoverlapped design such as the Micro/370, but it requires somewhat more logic.

Should Tredennick have used these alternative techniques in the Micro/370? Probably not. The slow control store is not seen to be a problem when it runs concurrently with a slow bus and a slow memory hierarchy. The 370 instruction set does not match a pipelined processor organization very well. Each of these techniques would require more chip space, which probably was not available. The trade-off choices made for the Micro/370 seem to form an effective combination, but there are many machine designs where they are not the right combination.

Design Process

There are two notable features of the way Tredennick goes about doing his design. First, his microprogramming syntax is graphic (flowcharts), using a two-dimensional positional notation with very concise variable names. This violates most of the rules of programming documentation because such a concise notation is difficult to learn or relearn. It is meaningful only to people deeply involved in the design. Note that his flowcharts contain no comments to help the reader understand the meaning of the actions taken. This is typical of designers who are totally immersed in the design process. Second, he uses virtually no computer aids. In fact, whenever the topic of computer-aided design (CAD) comes up, he runs hard in the other direction, rationalizing why CAD would be of no value.

His use of concise flowcharts probably is justified despite the poor user interface they present. The problem of programming a set of tightly synchronized concurrent activities makes traditional programming languages ineffective. Concise notations have great value in allowing the designer to see as much as possible of the design at one time.

Tredennick's avoidance of CAD is not as well justified because it could do him some good if he had the right programs. Computers can be valuable in storing, drawing, and reformatting flowcharts. With electronic flowcharting, the designer can rapidly assemble different perspectives on the microprogram, such as seeing all the states that use a particular register. Likewise, comments can be added to explain the logic to newcomers but can be kept out of the way when serious designers are at work. If this design work had been carried out a few years later, it probably would have used substantially more CAD tools.

Conclusions

Many of the techniques in this book are arguably useful only in the context of a microprocessor version of an IBM 370, but this design process has significance beyond that application. Virtually all microprocessor designs can benefit from the following views: that the controller is a state machine, with standard input/output/next-state relationships; that microprogramming of concurrent activities is the heart of the design process, particularly because the basic microinstructions can be tailored to achieve high program efficiency; and that flowcharts are an efficient way to bring together the large amount of information the designer must see at each design step.

The generic use of these ideas is perhaps best seen in the fact that I used almost exactly the same methods in a communications controller design in 1964. The technology was discrete transistors and magnetic core logic. If a design process can work over that wide a spread of function, time, and technology, it must hold elements of truth.

APPENDIX

A

The "Cultures" of Microprogramming

Four "cultures" have emerged in the world of microprogramming (figure A.1). I call them:

1. The commercial processor culture
2. The bit-slice culture
3. The microprogrammable processor culture
4. The single-chip culture

The commercial processor culture uses an "architecture" manual. IBM calls theirs the "principles of operation." It is the assembler language programmers' view and the prescribed view for processor designers. Enter microprogrammers, who program low-level hardware to give it the appearance of a processor with the instruction set in the architecture manual. Commercial processors are not meant to be microprogrammed by the people who buy them.

The bit-slice culture builds tailored processors out of standard bit-slice components. These processors usually are designed and microprogrammed by hardware experts; they usually are not meant to be microprogrammed by their ultimate users.

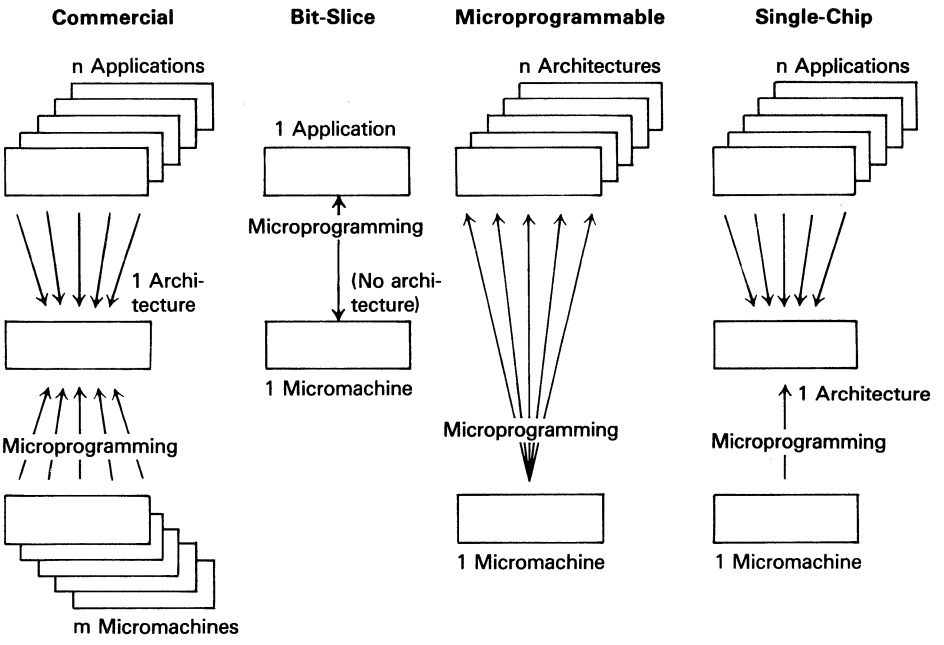


Figure A.1 The cultures of microprogramming

The microprogrammable processor culture designs processors that can emulate several computer architectures. Microprogrammable processors *are* meant to be microprogrammed by their buyers.

The single-chip culture focuses on individual chips—ones whose designs use microprogramming as an implementation technique, not as a feature. (This is a way to do it, not the neat thing about it.) These chips are not meant to be microprogrammed by the user. This is inside-the-chip microprogramming.

I will say more about cultures, then contrast some microprogramming folklore with realities from my culture (the single-chip culture). When you read or hear statements about microprogramming, know which culture is speaking. I hope some discrepancies between folklore and reality in the single-chip case will convince you that it's important to know the source. What's feasible for bit-slice can be crazy for single chips.

Microprogramming

> "Microprogramming" is just a term we misuse in common across four environments.

It's like saying "cooking" to a group of cooking fanatics. You convey some general image but not enough for a chef to know whether he or she cares about what you have to say. Microprogramming buffs do this with other terms: high-level language, microcode compaction, control store optimization, microcode verification, microprogrammable, micromachine, and more. Meanings depend on the environment in which people work.

The Commercial Processor Culture

IBM's System/370 is a good example of this culture. The *IBM System/370 Principles of Operation* prescribes software-compatible processors over a wide performance range. Most of the processors are microprogrammed.

Like all CPUs, a System/370 CPU has two parts: an execution unit (or data flow) and a controller. The execution unit contains

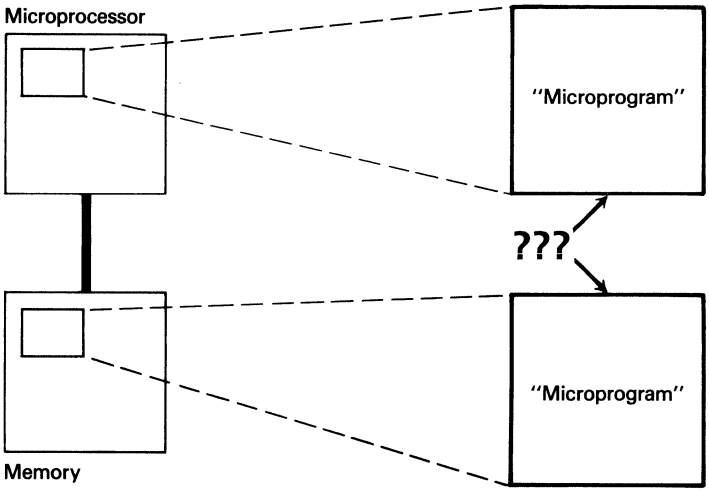


Figure A.2 Microprogramming anyone?

the general registers, ALU, shifter, PC, and so on—the resources needed to execute System/370 instructions (object code). The controller runs the execution unit (tells it what to do when). In a microprogrammed implementation, the controller consists of a control store (storage to hold microwords) and hardware that sequences the microwords. In System/370, this control store is a random access memory (RAM). IBM calls it reloadable control store (RCS) in order to distinguish it from the RAM used for main storage.

Implementations of System/370 CPUs are based on historical development. IBM engineers have been designing these CPUs for twenty-five years. They know how the block diagram of the CPU should look to attain set performance. When designing a particular model, engineers first draw a block diagram of a micromachine. They also publish a microinstruction manual for the micromachine. (Each CPU model is built on a different micro-machine so the manual is different for each one.) Microprogrammers then begin writing the programs—called microprograms—that run on the micromachine. Microprogrammers program the micromachine to make it look like the processor described in the principles of operation.

The microprogramming to implement System/370 is about half a million bits. Small processors have a simple micromachine and narrow microword but require many microwords. Big processors use a complicated micromachine with wider but fewer microwords. (The product of microword width and number of microwords stays about the same.) The control store usually is RAM and might even be pageable.

These processors often are called microprogrammable. After all, you can change the microprograms, add new instructions and diagnostic programs, and even provide tailored operating system assists. But if you program a System/370 micromachine to be something other than a System/370, say a Burroughs or Control Data architecture, it will not be very efficient. The micromachine is not designed to support just any architecture. It's designed to support one: System/370. When people from this culture say microprogrammable, they mean they can implement changes related to the architecture. This is a fixed-architecture microprogramming environment.

Microprogramming for these processors can be like programming in assembler language. The micromachine instructions (microcode) can be much like ordinary assembler-level instructions. These microprogrammed processors are believed to be slower than corresponding random logic or hardwired implementations. Why? Historically, the highest performance processors (the System/360 Model 91 and the System/370 Model 195, for example) were done in random logic.

The Bit-Slice Culture

Bit-slice processors use off-the-shelf chips (such as the Am2900 family) to build special-purpose programmed hardware. These processors are defined by the hardware designer to be used in a specific application; there is a custom micromachine for each application. There is no formal architecture document. The micromachine and the principles of operation level are the same, as there is no predefined, prescribed set of instructions to implement. The processor designer is also the microprogrammer. The processor usually is used for physical control applications. The designer uses the ones and zeros from the control store to manipulate external control lines (such as lights, wires, and switches) directly. The microprogramming to implement a bit-slice processor varies greatly. A simple engine controller might use only a few thousand bits; a sophisticated computer might use more than a million bits. Bit-slice processors may or may not be microprogrammed by their ultimate users. Usually, they are not. This is a bit-slice microprogramming environment.

The Microprogrammable Processor Culture

A third kind of microprogrammed processor is built like the micromachines in the commercial processor culture, but its execution unit is general purpose. This execution unit is meant to be used to emulate any architecture. The controller for a microprogrammable processor also is general purpose. The idea is to create a processor that can support any architecture just by changing the microcode. Changing control store contents (microcode) is easy. Microcode for these processors usually is paged because there can be so much of it. You might want to change

the architecture while the processor is running. (This is called dynamic microprogramming.) Programming for a microprogrammable processor might be supported by high-level languages, assemblers, linkers, compilers, and all the other ingredients from a "regular" programming environment. The Burroughs B1700 and the Nanodata QM1 are microprogrammable processors.

Because microprogrammable processors are versatile, they generate the most interest from people who want to develop microprogramming techniques. This is a variable-architecture microprogramming environment.

The Single-Chip Culture

A single-chip microprocessor such as the Motorola MC68000 is said to be a micropro*grammed* (not micropro*grammable*) processor. Like the commercial processors, the MC68000 has a "principles of operation," which Motorola calls a user's manual. Unlike the commercial processors, there is no microinstruction manual for the micromachine. The controller and execution unit for a single-chip microprocessor are designed to support a specific architecture. "The design might tolerate a different architecture by changing the microprogram; but the result can be like wearing a T-shirt upside down as underwear," says my collaborator, Brion Shimamoto. As with bit-slice processors, the hardware designer is also the microprogrammer, but the microprogram and the hardware are more closely related. The microcode is a result of the design procedure, not an independent thing. Unlike most bit-slice processors, there is a formal architecture. The user's manual prescribes which instructions are to be implemented. In direct contrast to processors in the microprogrammable processor culture, these are fixed architecture, nonmicroprogrammable processors.

"Single-chip" means that the control store, state sequencer, execution unit, bus controller, and clock-phase generators are on the same chip. The control store is ROM, which IBM calls read-only storage (ROS). Note: I do not count programming that resides in the main memory attached to a microprocessor as microprogramming. Microprogramming, in the context of single-chip microprocessors, is what resides in the on-chip control store. See figure A.2, page 247.

Contrasting Folklore and Reality

Because of the big investment (billions of dollars and thousands of man-years) in microprogramming, people want cheaper, more reliable ways of developing microprograms. Therefore, lots of people work on high-level languages, compilers, and assemblers for microprogramming, microprogram verification, control store optimization, and processor organization. Many people are developing theoretical bases and practical methods to reduce cost and improve reliability. This means there is too much to read. You need a fine filter—a finer filter than just the word "microprogramming."

I am from the single-chip culture. I worked at Motorola on the MC68000 microprocessor and at IBM on the Micro/370 microprocessor. My job was to design the microprogrammed controller, write the microprograms, and specify what went into the execution unit. I collected and read hundreds of pages on microprogramming, microprogrammed processor design, and microprogram optimization. I finally decided I couldn't use this material, set it aside, and did an empirical design. Now I know what happened. The articles were written for other microprogramming cultures—not mine. Here are some typical statements you might have seen in past and current literature. You may recognize them as part of the folklore of microprogramming. Here's how I think they apply to the single-chip culture.

Microprogramming is not very different from regular programming.

I would say the opposite—there is no relation between regular programming and microprogramming the controller for a single-chip microprocessor. For example, I design a controller using diagrams that combine state transitions and register transfers. These diagrams specify the exact operation of the processor. There is no programming in the "regular" sense of the word. The microword format is unknown until the diagrams are complete. Then a computer program generates microword bit patterns by a clerical translation of the diagrams. It still is not practical (it is possible) to use a high-level language to describe the operation of a commercial microprocessor.

One of the few problems of microprogramming that we can think of as solved is microcode compaction.

Compaction techniques are still not useful for reducing the size of the control store in *single-chip* microprocessors. (The control store is on the chip with the rest of the processor.) Current theory for reducing the number of bits in a microword depends on combining fields to control mutually exclusive functions. This idea assumes wires can cross freely, but in a single-chip implementation, wire crossings are not free. Combining controls for functions that are widely separated may use more area (to decode the signals and route the wires) than simply repeating the control field through the entire control store. To be useful, the theory must recognize the importance of signal position and the cost of wire crossings.

It is easy to correct design errors by making inexpensive changes in the microprogram, rather than expensive changes in the hardware.

In a single-chip implementation, one cost of the change is the same—you need a new chip. It might take longer to correct random logic than it takes to correct a microprogram, but even that isn't certain; it depends on how the random logic or microprograms were generated. A single-chip microprogrammed implementation is optimized to give maximum function per unit area for a specific architecture. Cross-microword dependencies can make microwords as tightly entwined as a set of reduced Boolean equations. Changing the microprogram can upset the whole thing. This is because the microwords were placed in the ROM by following a complicated procedure involving trade-offs among at least six rigid constraints. I believe that microword placement is a tougher problem than, say, an eight-variable Karnaugh map. (I've done both.) A small change in the microprogram can cause a ripple effect demanding that the placement procedure be repeated. This could be more expensive than changing a piece of random logic. It might not even be possible. Finally, the control store may not be large enough to stand the change.

Microprogramming permits the designer to put off decisions on the instruction set to be supported until the microprogrammable hardware has been almost finished.

This assertion is true, but changing the instruction set late in the design probably will increase the size of the design. For ex-

ample, the MC68000 microprocessor uses a type of two-level control store. Initially, we achieved a 2-to-1 ratio between the 17-bit microwords and the 68-bit nanowords, but late changes in the instruction set dropped the ratio to around 5-to-3. This cost about 3,500 bits of control store.

In a single-chip microprocessor, the execution unit is custom-tailored to the architecture (defined by the user's manual). Therefore, architecture changes may be difficult to support with existing hardware. A general-purpose execution unit would be too slow or too big. It would, for example, be difficult to add field manipulation instructions to an execution unit designed to manipulate only bits, bytes, and halfwords.

The largest drawback may be performance, because microprogrammed logic is generally slower than conventional logic.

The Zilog Z8000, the Intel 8086, and the Motorola MC68000 are all single-chip microprocessors of roughly equal technology. It would be difficult to predict which design was microprogrammed by measuring their performance at the pins with a logic analyzer. In this case, I think the microprogrammed processor is fastest, but we do not have microprogrammed and random logic implementations of the same architecture. In a single-chip microprocessor, the balance among the decoders, execution unit, and controller is crucial. If the delays through these elements can be properly balanced, it does not matter whether the controller is random logic or microprogrammed. In other words, the right internal structure contributes far more to performance than the choice of implementation method—random logic, microprogrammed, or programmable logic array.

Now that the cost of read/write memory is approaching that of read-only memory, it is likely that future microprogrammed computers will use ROMs only to protect small sections of control memory.

It is unlikely that single-chip microprocessors will be putting their microprograms in on-chip RAM because RAM density is too low to store a usable amount of microcode. Also, I want all the microprograms to reside on the chip permanently. Loading the microcode each time the chip is initialized is not efficient.

Paper Design	vs.	Real Design
Commercial vs. Bit-Slice Processor	vs.	Variable vs. Single-Chip Architecture
MicroproGRAMMABLE	vs.	MicroproGRAMMED
Industry	vs.	Academia
Electrical Engineering	vs.	Computer Science

Figure A.3 The table

Paper Design	vs.	**Real Design**
Commercial vs. Bit-Slice Processor	vs.	Variable vs. **Single-Chip** Architecture
MicroproGRAMMABLE	vs.	**MicroproGRAMMED**
Industry	vs.	Academia
Electrical Engineering	vs.	Computer Science

Figure A.4 The table (me)

Conclusion

Before you discuss microprogramming with someone, find out which culture they are in. (Maybe we should carry ID cards!) It saves time and frustration. Try using a table like the one shown in figure A.3. Circle your contexts and see if you can categorize the people to whom you talk. (My contexts are shown in figure A.4.) If you have trouble determining where a person or article is coming from, you are on the verge of wasting your time!

APPENDIX

B

Design Chronology

Sequence of Events (Overview)

The Micro/370 project began as something else. Three of us (Brion Shimamoto, Matt Halfant, and I) proposed a chip design project to management on 6 August 1980. It was tentatively approved. On 17 September 1980, we proposed that the chip be a communications chip, using IBM's Systems Network Architecture (SNA). We got another tentative approval. We began work. The SNA chip project was killed. About 17 December 1980, we looked at other chip candidates for implementation. We considered about ten candidates and picked System/370 on 7 January 1981. Matt was not allowed to join the project.

We did not have the resources to build a chip. Our initial objective (in a written objectives statement dated 15 December 1980) was "developing and documenting the design method for VLSI chips called the Flowchart Technique."

Architecture and Execution Unit

Brion started on the architecture of Micro/370 in January 1981. I began the design of Micro/370 with a proposed execution unit on 7 January 1981. From that time on, I modified the execution unit whenever the change made the chip faster or easier to

build. The execution unit went through about twenty major revisions. The initial execution unit had 16-bit macros. It had a single set of general registers, four arithmetic units, and no shifter. The final version has 32-bit macros with thirty-two general registers, two arithmetic units, and a single-cycle 64-bit shifter. These types of changes are expected and are a normal part of the flowchart method.

Flowcharts

I began working on flowcharts in January 1981. I started with practice flowcharts to get a feel for System/370 architecture. I was using the initially proposed execution unit and worked directly from the *IBM System/370 Principles of Operation* (form number GA22-7000). I worked on the flowcharts for about a year. I then collected the assumptions I had made in doing the flowcharts and defined the format for the control store word in January 1982. (Usually, the structure of the microcoded controller and the control word format are defined first. I let the flowcharts say what the controller structure and the control word format should be. This allows the processor requirements to determine the format for the control store word.)

Block Diagram and Floor Plan

Mon Yen Tsai (a circuit designer) and I talked about the chip floor plan in April 1982. Mon Yen and I did the floor plan for the Micro/370 chip in August 1982. The block diagram and the floor plan for Micro/370 are almost identical. This happened because I let the assumptions in the flowcharts determine the block diagram instead of choosing a block diagram at the start.

Simulation

To verify the design, we needed either a simulator or a hardware prototype. We chose simulation because we had a lot of computer time but no equipment or lab space. We had to have a flowchart assembler to generate the microcode bit patterns anyway, so we thought we could share some of the programs. We hired Linh Lam for the summer (between her school semesters) in May 1981.

In a characteristic underestimate of the work required, we thought she could do the flowchart simulator in three months. (She worked very hard, but could not save us from our estimate.) Linh's simulator verifies the flowcharts. It decodes

System/370 instructions, looks up the flowchart states, and simulates the register transfers in a programmed model of the execution unit. The flowchart simulator does not test the logic design of the control word decoders or the execution unit itself.

Dick Hadsell (a design automation engineer) joined the project in January 1983. He did not charge down the path where the essence of the design method being implemented takes a back seat to the incorporation of "innovative" ideas and to displays of programming skill. Instead, he studied the design method for a couple of months to see whether he was really interested in working with us.

Dick concluded that the flowchart method was worth supporting with computer tools. He didn't first try to convince me that his design automation tools were the right thing to do or that I should use them. He decided on objectives and began working on his own ideas. When he completed a program, he would ask me to try it and tell him what I thought.

Dick's first program was a flowchart assembler and syntax checker. The flowchart assembler takes the register transfer statements of the flowcharts and converts them into the ones and zeros of the control store. The syntax checker mainly says the flowchart assembler found something it could not convert into legal microcode. I began to use it immediately. There were numerous syntax errors in the flowcharts. When I added new microsequences or changed old ones, I ran the flowchart assembler as soon as I was done. I became addicted to it. It was easy to use and it did exactly what I needed—and no more. I didn't have to learn a new language or a new program interface.

Dick's second program was the control store placement program. This program uses the control store design constraints and places the control words in the control store. Micro/370 uses a two-level control store and a complicated ROM structure. The placement constraints are sufficiently complex that I assumed the placement would have to be done by hand. (I had done the placement for the Motorola MC68000 control store by hand.) Dick became interested in the problem and asked me to describe the method I used for control store placement in the MC68000. Then he asked me what was different about the placement constraints for Micro/370. Soon he had a control store placement program running.

Dick wrote a program to help me find all the paths through a given flowchart state. And a program to print the flowcharts from the data base in a format that suits my whims. And a program to sort and print categories of flowchart states.

Dick wrote FLASH, a general-purpose, interactive logic simulator, to test his ideas on what a logic simulator should have. He then modeled the Micro/370 design in FLASH. FLASH can run the entire Micro/370 chip, clock phase by clock phase. There is a simulated memory that allows you to watch Micro/370 execute programs. There is even a logic analyzer (square wave).

I wasn't pressured to use any of the tools, I didn't have to change my design method, and I didn't have to spend a lot of time learning how to use someone's idea of an "engineer's workstation." I used the tools because they performed specific functions I needed and because they fit the design method I use.

Logic Design

I began the logic design only after the (first pass) flowcharts were complete because the logic is defined by assumptions in the flowcharts. Until the flowcharts are complete, the logic isn't completely defined. When I thought the flowcharts were almost complete, I typed them into the computer. This served two purposes. First, I could use a text editor to find information (such as all the times I used PUR, for example). Second, this was the source data for the flowchart assembler and flowchart simulation program. I began the logic design in January 1982. The first PLAs I designed were the instruction decoders.

Circuit Design and Layout

Circuit design and layout for Micro/370 began when our first circuit designer joined the project. Mon Yen Tsai began working 20 percent time on Micro/370 in April 1981. At the time, I don't think Mon Yen could declare he was working on the project any more than that.

Most design activities could be concurrent. The architecture had to precede flowcharts, but flowcharts did not have to precede logic design. Logic design had to be ahead of the circuit design, and circuit design had to be ahead of layout. All of these things had to be done before the chip could be fabricated.

Overview of the Flowchart Method

The flowchart method is my empirically derived method. My background is electrical engineering: theory, semiconductors, and digital logic. I had experience in digital design and taught for a year before I went to work at Motorola. I was hired to work on the Motorola MC68000 microprocessor (not its name at the time). My job was to specify the controller and to do logic design. The flowchart method emerged from that experience.

Prerequisites and Objectives

To begin, you need three things: the instruction set specification (instruction definitions, formats, and address modes); the resources available to the programmer; and knowledge of the technology. The instruction set specification and the programmer resources are the architecture of the processor. They are defined in the *IBM System/370 Principles of Operation,* for the IBM System/370, and the user's manual for the Motorola MC68000. Knowledge of the technology comes from talking to the circuit designers.

You should have a documented design objective. A common objective is to make the processor as fast as possible. Another might be to make it fault tolerant. You must have at least one objective because it is the basis for choosing among design alternatives. If you have more than one objective, they must be ranked, as you cannot meet all objectives equally.

Flowcharts

The flowchart method is both procedure and notation. The designer reads the English specification of an instruction and, with reference to an execution unit model, writes register transfer statements to accomplish the instruction operations. These statements are called tasks. The statements follow a source-bus-destination notation. It doesn't get more complicated than that. The designer chooses the set of tasks he needs (making up the notation as he goes). An example task is dt→a→alu. This means put the contents of DT on the A bus to the ALU. Several tasks can occur simultaneously, if they use independent resources (such as different buses). A collection of one (really zero) or more tasks occurring simultaneously is called a state. A series of sequential states is called a sequence. Instructions

(those defined in the architecture) are accomplished using one or more state sequences. In a microcoded implementation, a state becomes a control word and a sequence becomes a control word sequence.

The flowchart method makes use of two kinds of flowcharts. In level 1 flowcharts, operation tasks and housekeeping tasks are listed in separate sequences. Operation tasks perform the required operand manipulations, such as a memory-to-register ADD. Housekeeping tasks are instruction overhead, such as address calculation, program counter updating, and prefetching. The idea is to find the most efficient operation sequences, changing the execution unit as necessary. In level 2 flowcharts, housekeeping tasks are merged into the operation sequences.

Implementation

You implement a design from a flowchart description by extracting assumptions from the flowcharts and then designing logic to support them. Suppose, for example, I found six different ways AT was used in the flowcharts. I would assign it 3 bits in the control word, assign the bit patterns, and design a decoder to activate the AT control lines.

Why It Works

I think the flowchart method works well because it does not impose a solution on the design. It is a systematic way to characterize the architecture so that design details emerge from the description. For a technology and set of design objectives, I think there is an implementation latent in the architecture. There is a processor that "wants" to be that architecture. It is "hidden" in the English description. The flowchart method helps find it. Using the flowchart method, the English description of the architecture is the designer's input. The designer's output is an execution unit (which evolves as the flowcharts develop) and the flowcharts themselves (which will become microcode in a microcoded implementation). From the flowcharts, the designer extracts information for the logic implementation—the instruction decoders, register selection logic, branch control, ALU and condition code control, and next state selection.

Chronology

Micro/370 was more of a grass-roots movement at IBM Watson Research than it was a regular project. The project started with two people, and it took two years to grow to seven or eight

people. Workwise, then, the rest of the project was way behind me when I decided to teach at the University of California–Berkeley for a year (1983). Later in 1983, the project took off. From 1983 to early 1986, there have been fifteen to twenty people (not the same fifteen to twenty people) on the project, not counting managers. In 1984, we made a Micro/370 test chip consisting of the execution unit, control word decoders, and clock-phase generators. In 1985, we made a full Micro/370 chip.

I kept two chronologies, one for design items and one for people items. I include the chronologies here to give you some live data on a real project. The chronologies stop in April 1983, when I took my sabbatical leave. It's just as well, as the number of people on board and things going on after that would have been too messy and too time-consuming to track. In all, from January 1981 to December 1986, the Micro/370 project survived ten second-level managers, six third-level managers, four fourth-level managers, and five office moves, in the computer sciences department alone. (Most of the physical design was done in a sister department.)

Micro/370 was an awfully slow-paced project. Being a grass-roots research project meant we were always severely bottlenecked by a lack of people and skills. A serious, all-out effort would have taken about two and a half years, max.

The numbers in parenthesis in the chronologies (figures B.1 through B.6) are the design revision levels of the execution unit and corresponding flowcharts at the time. The capitalized mnemonics are execution unit pieces or System/370 instruction mnemonics. The notation "computer files" refers to the entry of the design data into the computer. Before that, the data was kept on paper or vellum.

Earlier I said the project "took off" just about the time I went on sabbatical leave. A listing of all the people who contributed significantly to Micro/370 would have more than eighty entries. I haven't even met all of them! And listing the set of people who made significant contributions is tricky, at best, for a project as long and ambitious as this. I listed the people here who, I feel, contributed in some material way and somehow understood the focused, let's-do-it spirit of the Micro/370 project. This spirit, more than the raw technical contributions, made the experience worthwhile. Many others helped in a strictly technical or management sense, but these people are the ones who stick in my

mind. The people were from IBM locations in Endicott, New York; Essex Junction, Vermont; Los Gatos, California; Poughkeepsie, New York; and Yorktown Heights, New York.

The first group did the technical work, and for them I list the general area next to their names. The second group of people are the managers and staff who "greased the skids" for us. They opened doors and made things happen.

Technical

Bill Armstrong	Decimal instruction emulation code (student)
Daniel K. Beece	Switch-level simulation
Mark Birman	Circuit modeling, layout (student)
Michael Boguski	Performance modeling
Hu H. Chao	Circuit design, layout
Carol J. Chiang	Architecture verification (student)
Robert W. Donner	Circuit modeling
Victor Di Lonardo	Layout
Molly M. Elliott	Architecture verification
Bill B. Feaster	Circuit modeling, layout (student)
Bruce D. Gavril	Bus architecture
Paul F. Greier	Tektronix 3295 chip testing
Richard W. Hadsell	Logic verification
Donna P. Hawrot	Performance modeling
Joseph P. Higham	Layout, general information brochure
John C.L. Hou	Circuit design, layout
Leendert M. Huisman	Switch-level simulation
Vijay S. Iyengar	Switch-level simulation
Sivarama Kodukula	Card design
Linh H. Lam	Flowchart verification
Kelvin W. Lewis	Layout
Pete E. McCormick	Chip fabrication
Shauchi Ong	Logic design, circuit design, layout
Joyce H. Otis	Secretary
Stephen M. Parke	Circuit modeling, layout (student)
Thomas L. Schappe	Architecture verification
Robert G. Sheldon	Architecture verification

Feng-Hsien Warren Shih	Testing strategy, IBM LTE-240 chip testing, Tektronix 3295 chip testing
Brion Shimamoto	Micro/370 architecture
Gabby M. Silberman	Switch-level simulation (visiting scientist)
W. Bennett Smith	Circuit modeling, layout
Jeffrey Y. F. Tang	Logical-to-physical checking, layout
Nick Tredennick	Flowcharts, logic design, layout
Cynthia A. Trempel	Circuit design, layout
Mon Yen Tsai	Circuit design, layout
James W. Valashinas	Architecture verification
Frank Woytowich	IBM LTE-240 chip testing
David K. Yang	Circuit modeling, layout (student)

Managers and Staff

George S. Almasi
Robert W. Callahan
Richard P. Case
George J. Fan
Lucie J. Fjeldstad
Steve R. Houghtalen
Roland Y. Hung
E. Scott Kirkpatrick
Paul C. King
Daniel T. Ling
James F. McGroddy
Gavin Mills
William J. Nestork
Byron T. Rucker
Sue R. Sanicky
Herb Schorr
Jack R. Turnbull
Larry M. Warren
William W. White

Date—1981	Execution Unit	Flowcharts	Logic Design
January	R0-RF,ALU,AU,DI,DO, EX,PC,IRF,IRD,IRE (All were 16 bits.)	BCR,BC, address modes, LA,L, LR,LTR,LH,ST (Trial flowcharts)	Op code map
February	(1) 4-segment internal buses (2) AT,SFU	(Level 1) Address modes, BCR,BC,L,LR, LTR,LH,LA,ST,TM,MVC	
March	(3) DT	BALR,BAL,BCT,BCTR,BXH, BXLE,C,CL,CR,CLR,CLC,NC, OC,XC,CLI,CS,CDS,CH,STC, STH,STM	
April	(4) AU does A-B, ALU includes shifting (with DT on double-shift.)	LM,A,AL,ALR,AR,N,NR,O,OR, S,SL,SLR,SR,X,XR,SLA,SLL, SRA,SRL,SLDA,SLDL,SRDA, SRDL,NI,OI,XI,LNR,LPR,LCR,L, LR,LTR,LH,LA,LM,MVCL,CLCL, BCT (Level 2) Address modes, IC,BALR, BCTR,BAL,BCT,LA,BXH,LCR, LTR,BC,BCR,BXLE,LPR,LNR, LH,L,LR,LM	
May		CS,STC,STH,TM,CLC,STM,C, CL,CLR,CR,ST,SLDA,SLDR, SRDA,SRDR,SLA,SLR,SRA, SRR,CH,CLI,CDS,A,AL,ALR, AR,N,NR,O,OR,S,SL,SLR,SR,X, XR,NI,OI,XI,NC,OC,XC,AH,SH, MVC,MVI,MVCL,CLCL	
June	(5) 32-bit E-Unit (6) (7) SH,SL	(7) SLA,SLL,SRA,SRL,SLDA, SLDL,SRDA,SRDL,LPR,LNR	
July		M,MH,MR,D,DR,TM,LCR,LNR, LPR	
August	(8) PSWL,PSWH,ILC,AG	LPSW,BALR,BAL,ISK,SSK, SSM,SPM	
September	(9) PUR,DO (32 bit) (10)	(9) STIDP,IPK,TS,SPKA (10) MVCIN,TR,TRT,STNSM, STOSM,SVC	
October	(11)	(11) MVN,MVZ,PACK,UNPK, MVO,STCM,CLM,ICM	
November	(12) (13) (14) ACM,ACR (15)	(13) EX (14) CVB,CVD,DIAG (15) CP,AP,SP	
December	(16)	(16) Store-in-instruction stream, CS,STM,ST,CDS,MVCIN,NI,OI, XI,NC,OC,XC,MVC,MVI,TS,TR, MVCL,SSM,STIDP,STOSM, STNSM	DO,MXO,DI,MXI

Figure B.1 Design chronology 1981

Date—1982	Execution Unit	Flowcharts	Logic Design
January	(17)	(17) Exceptions, D,DR,LPSW, SVC,EX,DIAG,TM,IC,ST,SSM, STIDP,STOSM,STNSM,CVB	Op code map, A1,A2,PE, AG,SE,OE, microword format, nanoword format
February		BXH,BXLE,SLA,SLDA,SLDL, SLL,SRA,SRDA,SRDL,SRL, BALR,LM,CS,STM,CDS, MVCL,CLCL,M,MR,TRT,D, DR,LPSW,PACK,EX,CVB,CVD	Branch conditions, microword format, nanoword format
March	(18)	DIAG,STC,STH,UNPK,CVD	(18) Branch PLA, microword format, ALU&CC PLA
April	(19) PSWL (changed)		(19) Branch PLA, microword format, chip floor plan
June	(20)	(20) BXH,BXLE,LCR,LNR,LPR, CLCL,UNPK,CLM,ICM,CVB, CVD	(20) Branch PLA, microword format, nanoword format, ALU&CC PLA,SFU,PUR
July	(21) ACR deleted	(21) LCR,LTR,LNR,LPR,CVD, BASR,BCTR,BCT,LA,BC,BXH, LR,BXLE,SLA,SLDA,SLL,SLDL, SRA,SRDA,SRL,SRDL,BALR, LM,CS,C,CLC,TM,CL,CLR,CR, CH,CLI,IC,DIAG,AH,SH,MVCL, SPKA,TRT,STNSM,STOSM,D, DR,ICM,CLM,EX,NC,OC,XC, IPK,SPM,SVC,PACK,UNPK	(21) ALU&CC PLA,A1,AG, branch PLA, nanoword format
August	(22) ACM deleted, MXO deleted, PGH	(22) BAS,BXH,BXLE,BCR,BC, LH,CS,STH,CLC,AH,SH,CDS, TRT,STOSM,STNSM,STIDP, SVC,SPM,MVO,ICM,CL,EX,ST, address modes	(22) Controller block diagram, branch PLA, A1, A2, op code map, nanoword format, microword format, ALU&CC PLA
September	(23)	(23) STM,LNR,LPR,STIDP,TS	(23) PUR,RX,RY,AT, nanoword format, microword format, branch PLA, EDB,FC, ALU&CC PLA
	(24)	(24) MVCL,CLCL,STNSM, STOSM,D,DR,SVC,PACK,MVO	(24) CC,PCE,RX,RY,DI, SFU,ALU&CC,SPR, nanoword format, microword format
October		Address modes, SRA,SRDA, SLA,SLDA,L,M,CLR,CR,D, PACK,EX	Op code map, A1,A2,AG, ALU&CC,SFU, nanoword format, microword format, RX,RY,SPR
November	(25)	(25) Address modes, BCR,BCTR, SRDA,SPKA,SLA,SLDA,PACK, MVO,CVD	(25) Computer files: op code map, A1,A2, branch PLA, PE, nanoword format, microword format
December	(26) SFU placed above execution unit	(26)	(26) Op code map, A1, SPR, branch PLA, nanoword format, microword format, DI,DTI, AO,BMA,EDB,FC, interrupt priority

Figure B.2 Design chronology 1982

Date—1983	Execution Unit	Flowcharts	Logic Design
January	(27) Moved SH,SL,PUR	(27) Plotted FC, BCR,SRDA,CS, CDS,SLA,SLDA,STCM,CVB	(27) IFPO,PCR,FC
	(28) PGH (chg)	(28) BAL,BAS,CS,D,DR, interrupts, A,AL,ALR,AR,N,NR, O,OR,S,SL,SLR,SR,X,XR, STIDP,ICM,CLM,CVB,LPSW, DIAG,BCR,CVD,M,MR,MH, MVCL,PACK,STCM,SLDA,SLA, SRDA,SVC,CS,CDS, address modes, MVO,AH,SH,A,AR,S, SR,LPR,LCR,DIAG,SLDA,SLA	(28) Op code map, A1, microword, DI, nanoword, BC,SFU, shifter control, AUI,ALI,ALU&CC, shifter control
February		Pretesting added, TR,LM,STM, ICM,CLM,STCM,SVC,MVCL, CVD,CLCL,DIAG,ICPUR, exceptions, LPSW,IPK,SPKA, SSM,SPM,STOSM,STNSM, ISK,SSK,MVCL,PTLB,RRB	Computer files: PUR,AUI, ALI,PCE,SER,BMA,SFU, EDB,ALO,AO,AT,CCC, CCS,DI,DTI,IRD,MSB, RAB,SFB,ALU&CC
March		Floating-point coprocessor support, pretest reads, PACK, UNPK,MVC,NC,XC,OC,MVCIN, CLC,STCM, interrupts, SVC, CLCL,DIAG,IPK,D,DR	Floating-point coprocessor support, A1, A2,AG, computer files: SPR,AGB,ILC,PSW
April		IPL,ICPUR,CPUR, interrupts, RRB,PTLB,MVCIN	Computer files: A0,A0D, A0S
Once all the design data had been entered into computer files, there were no further editions of the flowcharts or the design notes on paper. The files were merely updated as needed and the revision number entered with each change. I could print copies of the flowcharts and PLAs or just look at the file at a terminal. (Edition numbers are in parentheses.)			

Figure B.3 Design chronology 1983

Date—1981	People	Events
January	Brion Shimamoto and Nick Tredennick begin Micro/370.	New second-line manager hired. Draw first execution unit. Begin flowcharts.
February		Outline chapters for this book. Finish top 10 instruction flowcharts. Nick applies for a sabbatical leave. Begin Level 1 flowcharts. New second-line manager hired. Nick becomes manager of Micro/370.
April	Mon Yen Tsai and Hu Chou start 20% time.	Begin Level 2 flowcharts. Begin flowchart-state 3x5 card file.
May	Linh Lam starts (summer hire). Linda Ebenstein starts (summer hire).	Begin layout. Begin flowchart verification program.
June		Mon Yen and Hu agree to work on Micro/370 until told to stop. Brion designs the Micro/370 logo. Order patches and baseball caps. Start recruiting Bruce Gavril.
July	Bruce Gavril starts. Linda Ebenstein is pulled off to another project.	Nick goes to IBM New Manager School. Brion and Nick have to move offices. Propose this book to Digital Press. Begin work on Micro/370 bus architecture—assume Intel 8086/88 bus compatibility.
August	Linh Lam returns to school.	Brion and Nick meet with Phyllis Keenan and Marcy Kenahl of Digital Press. Nick asks for three wafer runs starting in 4Q82. Private store concept is pitched out.
September		Propose Micro/370 instruction mix. First estimate performance 260 kips. Plot microwords, microcycles, kips, and bus utilization by instruction.
October		Bruce takes four weeks to move his office.
November		Begin working on decimal instructions. Baseball caps arrive. Finish studying Intel 8086/88 bus. Complete flowcharts for ninety-eight instructions.
December	Linh Lam returns for a month.	Pitch out decimal ops. Decide to allow store-in-instruction stream and byte-aligned operands. Finish studying Intel's Multibus. Fly swatters, hot pads, and decals all arrive. Nick begins typing flowcharts into the computer.

Figure B.4 Project chronology 1981

Date—1982	People	Events
January	Linh leaves again.	Nick finishes entering flowcharts. Begin logic design. Draw op code map. Define nanoword format.
February		Mon Yen told not to work on Micro/370.
March		Submit notice of intent to publish a book. Change to a 32-bit external bus. Sign a contract with Digital Press. New second-line manager hired. Decide Micro/370 will be Motorola MC68000 bus compatible, not Intel 808x bus compatible. Define microword format.
April		Decide to let people in IBM know about Micro/370. Determine execution time formulas.
May	Linh returns. Mon Yen starts 100% time.	Send contract to Digital Press.
June		New second-line manager hired. Change from 8- to 4-way micro branches in the control store. Brion gets a prolonged, recurring cold.
August	Linh leaves again. John Hou starts.	IBM Burlington promises to make wafers.
September	Naziko Asapova starts (temp).	Begin instruction decode simulator. Begin flowchart drawing program.
October		New second-line manager hired.
December	Linh returns. Naziko leaves.	Bruce completes initial specification of the bus architecture. Dick Hadsell talks to us about a job.

Figure B.5 Project chronology 1982

Date—1983	People	Events
January	Dick Hadsell starts part-time.	Add operand pretesting for virtual memory. Dick works on a flowchart assembler. Dick works on control store placement program. Dick gets flowchart drawing program working. Automatic flowchart-state 3x5 card file from Dick.
February		Compact control store. Flowcharts include operand pretesting.
March		Add trace, extra registers, Dual mode, coprocessor support, and other stuff. Bruce gets a correct suit. Logic design entered in computer.
April		Brion replaces Nick as manager of Micro/370.
May		Nick goes on sabbatical leave to teach for a year at the University of California-Berkeley.

Figure B.6 Project chronology 1983

A P P E N D I X

C

IBM Micro/370 Architecture Notes

Micro/370 is a 32-bit, single-chip, nMOS microprocessor. Its instruction set and operation are largely described by the manual *IBM System/370 Principles of Operation,* form no. GA22-7000-9. Micro/370 operates in the Extended Control (EC) mode of System/370.

Micro/370 directly executes 102 system/370 instructions. One of the 102 instructions, Diagnose, consists of fourteen varied and not necessarily related suboperations. The definition of Diagnose for Micro/370 is unique to Micro/370. (Diagnose, op code '83' hex, is model specific in System/370.) The operation of the remaining 101 instructions is as described by the *IBM System/370 Principles of Operation,* modified for the environment of 32-bit addressing.

Micro/370 is intended to be a component (not necessarily the central processing unit) of a system. From a programming view, Micro/370's close compatibility with System/370 (instructions, interruption handling, data formats) makes it practical for writing programs that run on Micro/370 today and will run on a larger System/370 engine tomorrow.

From a hardware view, Micro/370's bus is a superset of the Motorola MC68000 bus. Sixty-four of the 106 Micro/370 signal and

I/O pins are pin for pin compatible (electrically and protocol-wise) with the 64 pins of the Motorola MC68000. Engineers who are familiar with MC68000-based designs can be immediately familiar with Micro/370-based design. And MC68000 and MC6800 peripheral components can be used with Micro/370. You can design a system using Micro/370 and no other IBM parts. Micro/370 runs off a single 5-volt power supply. The chip requires an ordinary crystal oscillator for clocking. Micro/370 chips are designed to run at 10 MHz, worst case.

System/370 Instructions Not "On-Chip" in Micro/370

The following instructions are not implemented on-chip:

- Two of the general instructions: Monitor Call, Store Clock
- Three of the control instructions: Set Clock, Store Control, Load Control
- The eight I/O instructions
- The forty-four floating-point instructions
- The seven extended-precision floating-point instructions
- The Load Real Address instruction
- The four CPU timer and clock-comparator instructions
- The nine decimal instructions
- Thirty miscellaneous instructions comprising the following: Channel-Set Switching, Direct Control, Dual Address Space, IBM 3033 Extensions, Multiprocessing, Recovery Extensions, Storage-Key-Instruction Extensions, Suspend and Resume, System/370 Extended, and Test Block facilities.

In all, the above totals 107 instructions, which, if used, would have to be executed by coprocessors or simulated in software. Micro/370 implements coprocessor support for the fifty-one floating-point (regular and extended-precision) and the nine decimal instructions. Other instructions, including the I/O instructions, would be handled in Dual mode, a special processing mode of Micro/370. Dual mode, if enabled, causes the Micro/370 chip to execute instructions in a separate physical memory (called control space) whenever an interruption is taken. Control space programs are written in the 102 instructions directly executed by Micro/370. These programs have access to an extra set of sixteen general registers called shadow registers. Many of the Diagnose instruction suboperations (such as calculate effective address) are intended for use in Dual mode. Dual mode is described in appendix E.

Micro/370 Addresses

Micro/370 has a full 32-bit address bus. IBM System/370 architecture uses 24-bit addresses. This appendix describes how Micro/370 supports systems implementing either 24-bit or 32-bit addresses. Although I refer only to these two widths, you could implement any width between 24 and 32 bits by using the information for 32-bit addresses and ignoring the high-order bits. I refer to an IBM System/370 system with 24-bit addresses as a "System/370 system." I assume that it will be implemented with the help of the Dual mode feature; I refer to any program contained in control space as "control code."

Address Types in System/370

Micro/370 recognizes the following types of addresses in System/370: effective, virtual, and real. Since Micro/370 does not implement the System/370 Multiprocessing facility, a real address is the same to it as an absolute address. Since Micro/370 does not implement the System/370 Dual Address Space facility, Micro/370 does not itself make distinctions among the following types of System/370 addresses: logical, instruction, primary virtual, and secondary virtual.

Memory Addresses

Micro/370 always provides a full 32-bit address for memory accesses. Specifically, this applies to function codes 1, 2, 5, and 6 —that is, instructions and data in both 370 space and control space. A System/370 system must simply ignore bits A31–A24 of the address bus.

Program Status Word (PSW)

The instruction address in the PSW is a full 32-bit address (bits 32–63 of the PSW). A System/370 system must intervene in two ways to maintain a valid PSW:

1. Micro/370 allows nonzero bits in PSW bits 32–39. Therefore, control code must check the PSW in all situations that load a new PSW (Load PSW instruction, Diagnose–Dual Mode [Load PSW] instruction, or any PSW swap). If there are any ones in bits 32–39 of the new PSW, the control code must emulate a specification exception at the appropriate time, along with actions the System/370 architecture requires for the particular situation.
2. After every PSW swap, the control code must clear bits 32–39 of the old PSW.

Operand and Branch Addresses

The effective addresses of all operands are full 32-bit addresses. Since a System/370 system ignores bits A31–A24 for all memory accesses, this has no effect.

A branch address is the new instruction address when a branch instruction is executed and the branch is taken. All branch addresses are full 32-bit addresses. Since a System/370 system ignores bits A31–A24 for all memory accesses, this has no effect on it.

Dual-Mode Operation Exception and Privileged Operation Exception Handling

The calculated effective address(es) stored in general registers 1 and 2 are full 32-bit addresses.

Branch and Link Instructions (BAL, BALR)

This operates as defined for a System/370 system. For a 32-bit system, only the low-order 24 bits of the instruction address are contained in the link information. A program that may execute from addresses higher than 16M-bytes cannot use this instruction to establish a base register for addressing itself. Also, the link information cannot be used as a branch address in a 32-bit system because the high-order byte contains other information. The branch address is still a full 32-bit address.

Branch and Save Instructions (BAS, BASR)

This operates as defined for a System/370 system. For a 32-bit system, only the low-order 24 bits of the instruction address are contained in the link information. A program that may execute from addresses higher than 16M-bytes cannot use this instruction to establish a base register for addressing itself. Also, the link information cannot be used as a branch address in a 32-bit system if the original address may have been higher than 16M-bytes. The branch address is still a full 32-bit address.

Compare Logical Long Instruction (CLCL)

The operand addresses are full 32-bit addresses. After the operation completes, bits 0–7 of the addresses in the even-numbered general registers specified by the R1 and R2 fields are set to zero. (This is a System/370 requirement.) For a 32-bit system, the high-order byte of an operand address is lost if it was higher than 16M-bytes. If the operation is interrupted, bits 0–7 of the addresses in the even-numbered general registers

specified by the R1 and R2 fields are *not* set to zero. This preserves the full 32-bit operand addresses until completion. For a System/370 system, this is a deviation from the architecture; the control code that handles interruptions can correct it if necessary.

Diagnose—Calculate Effective Address(es)

The calculated effective address(es) stored at the second operand location are full 32-bit addresses.

Edit and Mark Instruction (EDMK)

The following description applies when a decimal coprocessor is attached. Otherwise, correct operation of EDMK is entirely the responsibility of the control code.

EDMK operates as defined for a System/370 system. For a 32-bit system, only the low-order 24 bits of the address of the first significant result byte are placed in general register 1. The high-order byte of the address must be reconstructed if it may have been higher than 16M-bytes.

Load Address Instruction (LA)

This operates as defined for a System/370 system. For a 32-bit system, only the low-order 24 bits of the effective address are placed in the register specified by the R1 field. LA cannot be used to calculate an address higher than 16M-bytes.

Move Long Instruction (MVCL)

The operand addresses are full 32-bit addresses. After the operation completes, bits 0–7 of the addresses in the even-numbered general registers specified by the R1 and R2 fields are set to zero. (This is a System/370 requirement.) For a 32-bit system, the high-order byte of an operand address is lost if it was higher than 16M-bytes. If the operation is interrupted, bits 0–7 of the addresses in the even-numbered general registers specified by the R1 and R2 fields are *not* set to zero. This preserves the full 32-bit operand addresses until completion. For a System/370 system, this is a deviation from the architecture; the control code that handles interruptions can correct it if necessary. In determining whether the operands overlap destructively, Micro/370 uses only the low-order 24 bits of the effective addresses and assumes that they wrap to zero at 16M-bytes. In this way, the instruction operates as defined for a System/370 system. For a 32-bit system, there are cases when the second operand crosses a 16M-byte boundary and when the in-

struction returns condition code 3 with no movement of data, although there was no real destructive overlap.

Translate and Test Instruction (TRT)

This operates as defined for a System/370 system. For a 32-bit system, only the low-order 24 bits of the argument address are placed in general register 1 when a nonzero function byte is found. The high-order byte of the address must be reconstructed if it may have been higher than 16M-bytes.

Interruptions

Micro/370 implements the six System/370 interruption classes on-chip. This means Micro/370 implements the following steps:

1. Detect occurrence.
2. Find the System/370 interruption class: restart, I/O, machine-check, program, external, or supervisor call. The interruption class implies a trap (PSW-swap) location, specified by System/370 architecture.
3. Check for interruption masking.
4. Determine interruption priority (specified by System/370 architecture).
5. If the interruption is allowed, perform a PSW swap according to System/370 architecture. If in Dual mode, perform the PSW swap in control space; if the exception is for operation or for privileged operation, handle according to the rules for Micro/370 Emulation Assist (Dual mode only).

Micro/370 handles Motorola MC68000 interrupts as System/370 I/O interruptions with a special interruption code. A Diagnose instruction sets the Motorola interrupt priority.

System/370 CPU Resets in Micro/370

System/370 architecture defines seven CPU resets. These are:

1. CPU power-on reset
2. Initial CPU reset
3. CPU reset
4. Clear reset
5. Initial program reset
6. Program reset
7. Subsystem reset

Micro/370 implements, via pins, initial CPU reset and CPU reset. The other resets, along with I/O system reset, can be constructed at the card level, without involving Micro/370.

APPENDIX

D

The Micro/370 Flowcharts

This appendix contains the flowcharts for the IBM Micro/370 microprocessor. They are not the final version, but they are very close. (The final version was not available in time for printing.) These flowcharts show the extent of the design and the level of detail in a commercial microprocessor project.

These flowcharts are useful for course assignments. You could redesign the control for the shifter, for example, using the information in these flowcharts. You could derive the instruction decoders, implement the branch control unit, derive control word formats, and place control words in the control store.

The flowcharts are a starting point for studying algorithms used in instructions like Multiply and Convert to Decimal. And you can study ways to speed execution.

rxh (ad1 → addr)

au→ao,pc dilz→b→au x2→a→au	irin
	0
	3
	a002
adrxh1 | | h	adrxh2
ag0→ad→dt au→ap→au edb→irf rb→b→au	frin
	0
	3
	a004
adrxh2 | |	adrxh3
alu→ad→sl au→b→ao,at edb→di pc→ap→au +2→au ls8 sh←sl←0	trop
	0
	2
	a006
adrxh3 | | h	ad2

rsm (ad1 → addr)

au→ao,pc dilz→a→at l1→sfu sfu→b→alu 0→alu	irin
	1
	a102
alsm01 | | h	alsm02
alu→ad→alu au→ap→au edb→irf sfu→bd→alu +2→au	frin
	4
	a104
alsm02 | |	alsm03
ag0f→ad→alu alu→bd→alu at→ap→au au→bp→at +3→au	nacc
	5
	a106
alsm03 | |	alsm04
alu→bd→sh au→bp→au rb→a→au ls2 sl←sh←0	nacc
	0
	a108
alsm04 | |	alsm05
au→ap→au sh→b→au sl→ad→sfu	nacc
	0
	1
	a110
alsm05 | |	ad2

rxw (ad1 → addr)

au→ao,pc dilz→b→au x2→a→au	irin
	0
	a202
adrxw1 | adrxh1 | h	adrxw2
ag0→ad→dt au→ap→au edb→irf rb→b→au	frin
	0
	a204
adrxw2 | adrxh2 |	adrxw3
au→a→ao,at,au di→b→dt,sh edb→di pswh→sfu +2→au	trop
	0
	3
	a206
adrxw3 | | w	adrxw4
ag0c→ad→dt au→ao dt→b→sh edb→dil pc→ap→au +2→au ls4 sh←sl←0	trop
	0
	a208
adrxw4 | | h	ad2

ssb (ad1 → addr)

ag0→ad→dt au→ap→ao,au dilz→b→at +2→au	irin
	0
	3
	a302
adssb1 | | w	adssb2
alu→dt au→ap→at,au dilz→b→alu edb→dil rb→a→alu +2→au	frin
	1
	1
	a304
adssb2 | |	adssb3
alu→a→at at→bp→ao sfu→bd→alu 0→alu	irin
	1
	a306
adssb3 | | h	adssb4
alu→dt au→pc dilz→b→au edb→di,irf rb→a→au	frin
	0
	a308
adssb4 | bxhrx2 |	ad2

ssbl (ad1 → addr)

au→ap→ao,au dilz→ad→alu sfu→b→alu,at,dt +2→au	irin
	1
	2
	a402
alssb1 | | h	alssb2
alu→*→au au→pc edb→di,irf mim→sfu rb→*→au	frin
	0
	4
	a404
alssb2 | |	alssb3
at→ap→au au→bp→ao,au dilz→*d→alu dt→*d→alu edb→dil	trow
	1s
	a406
alssb3 | | b	alssb4
alu→*→alu au→*p→at rb→*→alu	nacc
	1
	2
	a408
alssb4 | |	alssb5
alu→b→ao,au dt→a→au	irop
	0s
	a410
alssb5 | | b	alssb6
ag0→ad→sfu at→ap→au au→bp→at dt→bd→alu edb→dil 0→alu 0→au	frop
	1
	1
	a412
alssb6 | |	ad2

ssbi (ad1 → addr)

au→ap→ao,au dilz→ad→alu sfu→b→alu,at,dt +2→au	irin
	1
	a502
aissb1 | alssb1 | h	aissb2
alu→*→au au→pc edb→di,irf mim→sfu rb→*→au	frin
	0
	a504
aissb2 | alssb2 |	aissb3
at→ap→au au→bp→ao,au dilz→ad→alu dt→bd→alu edb→dil	trow
	4s
	a506
aissb3 | | b	aissb4
alu→*→alu au→*p→at rb→*→alu	nacc
	1
	a508
aissb4 | alssb4 |	aissb5
alu→b→ao,au dt→a→au	irop
	0
	1
	a510
aissb5 | | b	alssb6

rxb (ad1 → addr)

au→ao,pc dilz→b→au x2→a→au	irin
	0
	a010
adrxb1 | adrxh1 | h	adrxb2
ag0→ad→dt au→ap→au edb→irf rb→b→au	frin
	0
	a012
adrxb2 | adrxh2 |	adrxb3
alu→ad→sl au→b→ao,at edb→di pc→ap→au +2→au ls8 sh←sl←0	trop
	0
	a014
adrxb3 | adrxh3 | b	ad2

sib (ad1 → addr)

au→ao,pc dilz→a→au 0→au	irin
	0
	a114
adsib1 | | h	adrxb2

rsh (ad1 → addr)

ag03→ad→sfu au→ap→ao,au +2→au	irin
	0
	a212
adrsh1 | | h	adrsh2
au→pc dilz→*→alu,au edb→di,irf rb→*→alu,au	frin
	1
	1
	a214
adrsh2 | |	ad2

rxs (ad1 → addr)

au→pc dilz→*→au x2→*→au	nacc
	0
	4
	a312
adstr1 | |	adstr2
ag0f→ad→sfu au→ap→au rb→b→au	nacc
	0
	a314
adstr2 | basxw2 |	ad2

rr — ad1 — bcr

au→pc irf→ird r2→a→ao,au +2→au	irin
	0
	b002
bcrrw1 | | h	bcrrw3
ag01→a→sfu edb→di,irf sfu→b→sh rs16 0→sh→sl	frin
	0
	b004
bcrrw3 | ipks02 |	bc00

cc = true → bctrw3
cc = false → bcrrw2

rr — ad1 — bcr(nop)

au→a→ao,au,pc di→b→alu irf→ird 0→alu +2→au	irin
	1
	2
	b102
bcnop1 | | |	bcnop2
edb→di,irf	frin
	0
	1
	b104
bcnop2 | | |	ad1

rr (r2≠0) — ad1 — bctr

au→pc irf→ird r1→b→alu r2→a→ao,au −1→alu +2→au	irin
	1
	b202
bctrw1 | | h	bctrw2
alu→*→r1 edb→di,irf r2→*→at	frin
	0
	b204
bctrw2 | bctrz2 |	bc14

alu = 0 → bcrrw2
alu ≠ 0 → bctrw3

rr — ad1 — basr

au→a→ao,au,pc irf→ird pce→bz→r1 +2→au	irin
	0
	b302
basrz1 | | h	bcnop2

rs — ad1 — bxh

au→ap→ao,au r1→ad→alu r2→b→alu +2→au	irin
	1
	1
	b402
bxhrx1 | | h	bxhrx2
alu→dt au→pc dilz→b→au edb→di,irf rb→a→au	frin
	0
	6
	b404
bxhrx2 | |	bxhrx3
alu→b→alu,sh au→ap→ao,au irf→ird r2(1)→ad→alu +2→au	irin
	3
	1
	b406
bxhrx3 | | h	bxhrx4
edb→di,irf sh→b→r1	frin
	0
	b408
bxhrx4 | slrrs2 |	bc03

aluv = 0 → bxhrx5
aluv = 1 → bxhrx6

rx — ad1 — bal

au→pc dilz→*→au x2→*→au	nacc
	0
	b502
balxw1 | adstr1 |	balxw2
ag0f→ad→sfu au→ap→au rb→b→au	nacc
	0
	b504
balxw2 | basxw2 |	balxw3
au→b→ao,at bc→sfu irf→ird pce→a7→sfu	irin
	0
	b506
balxw3 | balrz1 | w	balxw4
at→a→au,pc edb→di,irf sfu→b→r1 +2→au	frin
	0
	b508
balxw4 | balrz2 |	bctrw3

edb→di,irf
pc→a→ao,au
+2→au

trin
0
2
b007
bcrrw2 | | h | ad1

rx — ad1 — bct

au→ap→ao,au dilz→*→alu x2→*→alu +2→au	irin
	1
	1
	b107
bctrx1 | | h	bctrx2
alu→*→au au→pc edb→di,irf mim→sfu rb→*→au	frin
	0
	b109
bctrx2 | alssb2 |	bctrx3
au→a→ao,au irf→ird r1→b→alu −1→alu +2→au	irin
	1
	b111
bctrx3 | | h	bctrw2

au→a→ao,au,pc
irf→ird
+2→au
0xxxxxx→etmssrr

irin
0
b114
bctrw3 | | h | bcnop2

rr (r2≠0) — ad1 — balr

edb→irf pce→b→dt r2→a→ao,au,pc +2→au	trin
	0
	b207
balrw1 | | w	balrw2
au→b→ao,at bc→sfu dt→a7→sfu irf→ird 0xxxxxx→etmssrr	irin
	0
	b209
balrw2 | | h	balrz2

rr (r2=0) — ad1 — bctr

au→a→ao,au,pc irf→ird r1→b→alu −1→alu +2→au	irin
	1
	b212
bctrz1 | | h	bctrz2
alu→*→r1 edb→di,irf r2→*→at	frin
	0
	2
	b214
bctrz2 | | |	ad1

ag3f00→a→alu
alu→dt
sh→b→alu

nacc
5
b305
bxhrx6 | spmrr3 | bc02

alu < 0 → bcrrw2
alu ≥ 0 → bctrw3

rx — ad1 — bc

au→ap→ao,au dilz→*→alu x2→*→alu +2→au	irin
	1
	b309
bcrxw1 | bctrx1 | h	bcrxw2
alu→*→au au→pc edb→di,irf mim→sfu rb→*→au	frin
	0
	b311
bcrxw2 | alssb2 |	bc00

cc = true → bcrxw3
cc = false → mims05

au→a→ao,au
ipb→sfu
edb→di,irf
+2→au
rs32 0→sh→sl

trin
0
b314
bcrxw3 | exrx03 | w | bctrw3

ag0→ad→sfu

nacc
0
b411
bxhrx5 | cpur05 | bc02

alu < 0 → bctrw3
alu ≥ 0 → bcrrw2

rr (r2≠0) — ad1 — basr

edb→irf pce→bz→r1 r2→a→ao,au,pc +2→au	trin
	0
	b414
basrw1 | | w | bctrw3

rr (r2=0) — ad1 — balr

au→b→ao,at bc→sfu irf→ird pce→a7→sfu	irin
	0
	1
	b512
balrz1 | | h	balrz2
at→a→au,pc edb→di,irf sfu→b→r1 +2→au	frin
	0
	1
	b514
balrz2 | | |	ad1

rx — ad1 — bas

au→pc dilz→*→au x2→*→au	nacc
	0
	b010
basxw1 | adstr1 |	basxw2
ag0f→ad→sfu au→ap→au rb→b→au	nacc
	0
	3
	b012
basxw2 | |	basxw3
au→a→ao,au,pc edb→irf pce→bz→r1 +2→au	trin
	0
	b014
basxw3 | | w	bctrw3

rs — ad2 — cs

ag07→b→alu au→a→alu,sl mim→sfu	nacc
	5
	2
	c002
csrsw1	csrsw7
wait	nacc
	0
	c004
csrsw7 ddr11	bc28

sl(30:31) = 0 — csrsw8
sl(30:31) ≠ 0 — spexc1

au→ap→ao,at,au edb→di +2→au	trop
	0
	c007
csrsw8 trss13 w	csrsw9
au→ao edb→dil	trop
	0
	c009
csrsw9 cdsr06 h	csrsw0
di→b→alu irf→ird pc→a→ao,au r1→ad→alu +2→au	irin
	3
	c011
csrsw0 h	csrsw2
at→ap→au di→ad→dt edb→di,irf r2→b→alu 0→alu 0→au	frin
	1
	c013
csrsw2	bc14

alu ≠ 0 — csrsw3
alu = 0 — csrsw4

rs — ad1 — set trace diag.

x1xxxxx→etmssrr	nacc
	0
	1
	c102
trace1	dpsw01

edb→di,irf pc→ap→ao,au sh→b→alu sl→ad→alu +2→au	trin
	2
	c105
clcss6 h	ad1

alu→*→do au→a→ao,au doh→edb +2→au	wrop
	0
	6
	c108
csrsw4 w	csrsw5
alu→*→do au→ao dol→edb pc→a→au −2→au	wrop
	0
	c110
csrsw5 h	csrsw6
au→a→ao,au edb→irf ex→b→alu +4→alu +2→au	trin
	1
	c112
csrsw6 mvc113 w	bcnop1
dt→*→r1 pc→a→au +2→au	nacc
	0
	1
	c114
csrsw3	ad1

ss — ad2 — clc

alu→*→alu au→ap→ao,au dt→*→alu edb→dil irf→ird +1→au	trop
	4
	c202
clcss1 b	bc14

alu = 0 — clcss4
alu ≠ 0 — clcss2

at→ap→ao,au au→bp→at di→ad→dt dt→bd→alu edb→dil −1→alu +1→au	trop
	1
	1
	c205
clcss2 b	bc20

alub = 0 — clcss3
alub ≠ 0 — clcss6

alu→dt at→ap→ao,au au→bp→at di→ad→alu,sl dt→bd→alu,sh edb→dil +1→au	trop
	4
	c208
clcss3 b	bc14

alu = 0 — clcss4
alu ≠ 0 — clcss2

at→ap→ao,au au→bp→at dl→ad→dt dt→bd→alu edb→dil −1→alu +1→au	trop
	1
	c211
clcss4 clcss2 b	bc20

alub = 0 — clcss5
alub ≠ 0 — clcss6

di→ad→alu dt→b→alu edb→di,irf pc→ap→ao,au +2→au	trin
	2
	c214
clcss5 h	ad1

rs — ad2 — cds

ag07→b→alu au→a→alu,sl mim→sfu	nacc
	5
	c302
cdsr01 csrsw1	cdsr02
di→b→sh sfu→bsb rs8 0→sh→sl	nacc
	0
	c304
cdsr02 mclin3	bc14

alu = 0 — cdsr03
alu ≠ 0 — spexc1

au→a→ao,au di→b→dt,sh edb→di +2→au	trop
	0
	3
	c307
cdsr03 w	cdsr04
au→a→ao,au edb→dil r1→b1→dt +2→au rs32 0→sh→sl	trop
	0
	3
	c309
cdsr04 h	cdsr05
au→a→ao,at,au di→b→dt,sh edb→di pswh→sfu +2→au	trop
	0
	c311
cdsr05 adrxw3 w	cdsr06
au→ao edb→dil	trop
	0
	5
	c313
cdsr06 h	cdsr07

dt→*→alu irf→ird pc→ap→ao,au r1→*→alu +2→au	irin
	3
	c401
cdsr07 h	cdsr08
di→b→alu,sh edb→di,irf r1(1)→a→alu	frin
	4
	c403
cdsr08	bc14

alu ≠ 0 — cdsr10
alu = 0 — cdsr09

r1(1)→a→alu sh→b→alu	nacc
	3
	c406
cdsr09	bc14

alu ≠ 0 — cdsr10
alu = 0 — cdsr11

at→ap→ao,au doh→edb r2→b→alu r2(1)→ad→do 0→alu +2→au	wrop
	1
	c409
cdsr11 w	cdsr12
at→a→au au→ao dol→edb r2(1)→b→do −4→au	wrop
	0
	c411
cdsr12 h	csrsw4

dt→a→r1 sh→b→r1(1)	nacc
	0
	c414
cdsr10	ad1

rr — ad1 — clr,cr

au→ap→ao,au,pc irf→ird r1→ad→alu r2→b→alu +2→au	irin
	2
	2
	c502
cclrr1 h	bcnop2

rx — ad2 — ch

au→ap→ao,au,pc dile→b→alu irf→ird r1→ad→alu +2→au	irin
	2
	1
	c505
chrxh1 h	bcnop2

si — ad2 — cli

au→ap→ao,au,pc di→ad→alu irf→ird sfu→b→alu +2→au	irin
	2
	c508
clisi1 tmrxb1 h	bcnop2

rx — ad2 — c,cl

au→ap→ao,au,pc di→b→alu irf→ird r1→ad→alu +2→au	irin
	2
	2
	c511
cclrx1 h	bcnop2

rr
ad1
dr
au→pc
irf→ird
r1→b→alu,sh
r2→a→at,au,sfu
0→alu
0→au
nacc
1
d002
ddr01
ddr02
ag0→b→alu,au
alu→dt
au→ap→au
r1(1)→ad→alu,sl
nacc
2s
d004
ddr02
bc09
(alu<0)(au≥0) ddr08
(alu≥0)(au<0) ddr03
(alu≥0)(au≥0) ddr09
(alu<0)(au<0) ddr07
rx
ad2
d
au→pc
di→a→at,au,sfu
irf→ird
r1→b→alu,sh
0→alu
0→au
nacc
1
d102
ddrx01
ddr02
at→a→alu
sh→b→alu,dt
nacc
2
1
d105
ddr09
ddr04
au→a→alu,at
sh→b→alu,dt
nacc
2
d202
ddr03
ddr04
ag20→a→au
dt→b→sh
−1→au
ls1 sh←sl←0
nacc
0
d204
ddr04
bc24
aluc=1 ddr23
aluc=0 ddr14
alu→a→sl
au→ap→at
dt→b→alu,sh
−1→alu
nacc
4
d302
ddr07
bc24
aluc=1 ddr06
aluc=0 ddr05
alu→b→alu
+1→alu
nacc
1
d305
ddr06
stcm12
ddr05
alu→a→sl
dt→b→alu,sh
−1→alu
nacc
4
d402
ddr08
bc24
aluc=1 ddr06
aluc=0 ddr05
alu→b→alu,dt,sh
at→a→alu
nacc
2
d405
ddr05
ddr04
rs
ad2
slda
irf→ird
pc→ap→ao,au
r1→b→alu,sh
r1(1)→ad→alu,sl
+2→au
shift
irin
1
d502
sldas1
srdas1
h
bc04
pswh (20)=0 sldas2
pswh (20)=1 sldas3
ag0→b→alu
edb→di,irf
sl→a→alu,r1(1)
frin
2
d505
sldas2
srdas2
bc08
ovf=1 slars3
ovf=0 srdas3
au→a→au
dt→b→sh
−1→au
ls1 sh←sl←0
nacc
0
d008
ddr12
bc29
au(24:31)=0 ddr15
au(24:31)≠0 ddr14
alu→b→dt,sh
alu→a→au
−1→au
ls1 sh←sl←1
nacc
0
d108
ddr13
bc29
au(24:31)=0 ddr15
au(24:31)≠0 ddr14
ag0c→a→au
alu→dt
au→b→alu
0→alu
+4→au
nacc
1
d208
ddr19
bc13
au<0 ddr23
au≥0 ddr22
alu→a→dt
au→b→alu
0→alu
nacc
4
d308
ddr18
ddr21
cpur
a01
cpu reset
au→pc
ilc→sfu
xxx01xx→etmssrr
nacc
0
d408
cpur01
sosdr1
bc10
pswh (14)=0 cpur02
pswh (14)=1 cpur05
ag0→b→alu
edb→di,irf
sl→a→alu,r1(1)
frin
2
d508
sldas3
srdas2
bc08
ovf=1 sldas4
ovf=0 srdas3
at→a→alu
sh→b→alu,dt
nacc
2
d011
ddr14
ddr09
ddr11
wait
nacc
0
3
d013
ddr11
bc24
aluc=1 ddr13
aluc=0 ddr12
r1→a→au
sfu→b→alu
0→alu
0→au
nacc
1
d111
ddr15
ddr16
dt→b→alu
sl→a→au
0→alu
0→au
nacc
4
d113
ddr16
bc09
(alu≥0)(au≥0) ddr17
(alu<0)(au≥0) ddr20
(alu≥0)(au<0) ddr18
(alu<0)(au<0) ddr19
au→b→alu
dt→a→au
0→alu
0→au
nacc
1
1
d211
ddr17
bc13
au<0 ddr23
au≥0 ddr22
alu→ad→r1(1)
dt→b→r1
edb→di,irf
pc→ap→ao,au
+2→au
trin
0
d214
ddr22
h
ad1
au→b→alu
0→alu
nacc
4
1
d311
ddr20
ddr21
alu→*→au
dt→*→au
nacc
0
d313
ddr21
apssb6
bc05
alu>0 ddr23
alu≤0 ddr22
alu→dt
pswl→b→alu
−4→alu
ls8 sh←0
nacc
1
d411
cpur02
spexc2
cpur03
alu→a→au,sl
sfu→b→au
0011101→etmssrr
nacc
0
d413
cpur03
cpur04
stop
a0
/370 stop
ag0→ad→sfu
nacc
0
1
d511
cpur05
cpur02
au→pc
r1(1)→az→alu,au
r2→b→alu,at
0→au
nacc
1
d514
cpur04
stop01

rx (r1 ≠ 0) ad1 ex
au→pc dilz→*→au x2→*→au nacc 0 e002
exrx04 adstr1 exrx05
ag440000→bd→dt,sh au→bp→au rb→a→au rs8 0→sh→sl nacc 0 e004
exrx05 exrx02 exrx06
au→a→ao,au edb→di,irf r1→b1→dt +2→au rs32 0→sh→sl trin 0 e006
exrx06 w exrx07
rx (r1 = 0) ad1 ex
au→pc dilz→*→au x2→*→au nacc 0 e102
exrx01 adstr1 exrx02
ag440000→bd→dt,sh au→bp→au rb→a→au rs8 0→sh→sl nacc 0 1 e104
exrx02 exrx03
au→a→ao,au ipb→sfu edb→di,irf +2→au rs32 0→sh→sl trin 0 2 e106
exrx03 w exrx07
dile→bd→alu,sh dt→a→alu pc→bp→ex rs8 0→sh→sl nacc 3 e108
exrx07 exrx08
alu→dt au→ap→ao,au,pc sh→b→alu sl→ad→alu +2→au irin 4 e110
exrx08 h exrx09
dilz→b→at dt→a→sfu edb→dil irf→ird frin 0 e112
exrx09 bc20
alub = 0 exec1
alub ≠ 0 exrx10
none db execute exception
ag03→a→sl pswh→sfu pswl→b→alu −4→alu 0011101→etmssrr nacc 1 e202
exec1 opexc2
none db fixed point overflow
ag08→a→sl pswh→sfu pswl→b→alu −4→alu 0011101→etmssrr nacc 1 e205
fpovel opexc2
none db data exception
ag07→a→sl pswh→sfu pswl→b→alu −4→alu 0011101→etmssrr nacc 1 e208
dtexc1 opexc2
none db fixed divide overflow
ag09→a→sl pswh→sfu pswl→b→alu −4→alu 0011101→etmssrr nacc 1 e211
ddr23 opexc2
int a01 check stop
ag06→a→sl pswh→sfu 0011101→etmssrr halt 0 e214
chkst1 spexc1 ad5
icpur a01 initial cpu reset
ag80000→a→at,sfu au→pc xxx10xx→etmssrr nacc 1 e302
icpur1 sosdr2 icpur2
ag0→a→au,do,pc at→b→ao doh→edb sfu→mim 0→au swak 0 e304
icpur2 icpur3
at→b→ao au→a→do,sl dol→edb sfu→pswh 0011101→etmssrr swak 0 e306
icpur3 stop01
none a01 specification exception
ag06→a→sl pswh→sfu 0011101→etmssrr nacc 0 1 e309
spexc1 spexc2
alu→dt pswl→b→alu −4→alu ls8 sh←0 nacc 1 7 e311
spexc2 opexc2
none a04 /370 stopped
pc→a→au +4→au nacc 0 2 e314
stop01 ad4
int a0 eudump
ag100→a→ao,au doh→edb dt→b→do +2→au wrro 0 e402
eudu00 w eudu02
ag28→a→ao,at,au doh→edb sfu→b→do +2→au wrro 0 1 e405
opexc2 w opexc3
ag8c→ad→dt au→ap→ao,au dol→edb ilc→sfu sfu→b→do +2→au wrro 0 e407
opexc3 mvisi3 h opexc4
alu→a→alu au→pc sfu→b→alu xxx11xx→etmssrr nacc 1 e409
opexc4 sosdr3 prin04
rs ad2 sra
irf→ird pc→ap→ao,au r1→b→alu,sh r1(1)→ad→alu,sl +2→au shift irin 1 e412
srars1 srdas1 h srars2
edb→di,irf sh→b→alu,r1 0→alu frin 2 2 e414
srars2 ad1
wait ad3 wait
pc→a→au +4→au nacc 0 e502
wait01 stop01 ad3
rx a0 ex
edb→irf ex→a→ao,au +2→au trin 0 e505
exrx12 h bctrw3
none ad1 privileged operation
agba→a→au msb→sfu +6→au nacc 0 e508
popex1 opexc1 popex2
start a0 /370 start
pc→a→au +4→au nacc 0 e511
start0 stop01 bc10
pswh(14) = 0 start1
pswh(14) = 1 wait01
edb→di,irf pc→a→ao,au +2→au trin 0 e514
start1 bcrrw2 w bcnop1
di→b→pur dt→a→pur sfu→ird nacc 0 e010
exrx10 mvzss3 exrx13
dt→a→alu sh→b→alu nacc 4 e012
exrx13 aussb4 exrx11
1xxxxxx→etmssrr nacc 0 e014
exrx11 ad6

rx — ad2 — al, n, o, sl, x

au→ap→ao,au,pc di→b→alu irf→ird r1→ad→alu +2→au			irin
			2
			f002
oprrx1	cclrx1	h	bctrz2

si — ad2 — ni, oi, xi

di→a→alu sfu→b→alu			nacc
			2
			f005
noxsi1			noxsi2
alu→b→do at→*p→ao dol→edb sl→a→sfu			wrop
			0
			1
			f007
noxsi2		b	start1

rx — ad2 — ah, sh

au→ap→ao,au,pc dile→b→alu irf→ird r1→ad→alu +2→au			irin
			2
			f010
ashrx1	chrxh1	h	bc04

pswh(20) = 0 **bctrz2**
pswh(20) = 1 **ashrx2**

alu→*→r1 edb→di,irf r2→*→at			frin
			0
			f013
ashrx2	bctrz2		bc03

aluv = 0 **slars4**
aluv = 1 **fpove1**

ss — ad2 — nc, oc, xc

au→ao edb→dil			trow
			0
			f102
noxss1	cdsr06	b	noxss2
at→a→ao,au au→b→at di→*d→dt edb→dil +1→au			trop
			0
			f104
noxss2	mvcss1	b	noxss4
alu→dt di→b→alu dt→a→alu			nacc
			2
			f106
noxss4			bc14

alu = 0 **noxss6**
alu ≠ 0 **noxss3**

alu→ad→do at→ap→ao,au au→bp→at dol→edb dt→bd→alu −1→alu +1→au			wrop
			1
			1
			f109
noxss3		b	bc20

alub = 0 **noxss1**
alub ≠ 0 **noxss5**

rr — ad1 — ar, sr

au→ap→ao,au,pc irf→ird r1→ad→alu r2→b→alu +2→au			irin
			2
			f113
asrrr1	cclrr1	h	bc04

pswh(20) = 0 **bctrz2**
pswh(20) = 1 **ashrx2**

rr — ad1 — mr

au→ap→ao,au,pc r1(1)→ad→sl r2→b→alu,dt 0→alu +2→au			irin
			5
			f202
mulsw1		h	mulsw2
ag20→a→au alu→b→sh edb→di,irf irf→ird −1→au			frin
			0
			f204
mulsw2			bc16

sl(31) = 0 **mulsw4**
sl(31) = 1 **mulsw5**

alu→ad→do at→ap→ao,au au→bp→at dol→edb dt→bd→alu −1→alu +1→au			wrop
			1
			f207
noxss6	noxss3	b	bc20

alub = 0 **cead10**
alub ≠ 0 **start1**

ag01→a→sfu au→ao edb→dil			trow
			0
			1
			f210
noxss5		b	noxss2

rr — ad1 — alr, nr, or, slr, xr

au→ap→ao,au,pc irf→ird r1→ad→alu r2→b→alu +2→au			irin
			2
			f214
oprrr1	cclrr1	h	bctrz2

rx — ad2 — m

au→ap→ao,au,pc di→b→alu,dt r1(1)→ad→sl 0→alu +2→au			irin
			5
			f302
mulsx1		h	mulsw2
sh→b→alu 0→alu msb→sfu			nacc
			1
			3
			f304
mulsw3			mulsw4
alu→b→sh au→a→au −1→au rs1 n.eor.v→sh→sl			nacc
			0
			1
			f306
mulsw4			bc31

(au(24:31) ≠ 0) **mulsw3**
(sl(30:31) = 00, 11)
(au(24:31) ≠ 0) **mulsw5**
(sl(30:31) = 10)
(au(24:31) ≠ 0) **mulsw6**
(sl(30:31) = 01)
au(24:31) = 0 **mulsw7**

dt→a→alu sh→b→alu			nacc
			2
			1
			f310
mulsw5			mulsw4
dt→a→alu sh→b→alu			nacc
			1
			1
			f312
mulsw6			mulsw4
pc→ap→au sh→b→r1 sl→ad→r1(1) +2→au			nacc
			0
			f314
mulsw7			ad1

rx — ad2 — mh

au→ap→ao,au,pc dile→b→alu,sh r1→ad→dt 0→alu +2→au			irin
			5
			f402
mulsh1		h	mulsh2
ag10→a→au edb→di,irf irf→ird −1→au rs32 0→sh→sl			frin
			0
			f404
mulsh2			mulsh9
r1→b→at			nacc
			0
			f406
mulsh9			bc16

sl(31) = 0 **mulsh4**
sl(31) = 1 **mulsh5**

sh→b→alu 0→alu msb→sfu			nacc
			1
			f409
mulsh3	mulsw3		mulsh4
alu→b→sh au→a→au −1→au rs1 n.eor.v→sh→sl			nacc
			0
			f411
mulsh4	mulsw4		bc31

(au(24:31) ≠ 0) **mulsh3**
(sl(30:31) = 00,11)
(au(24:31) ≠ 0) **mulsh5**
(sl(30:31) = 10)
(au(24:31) ≠ 0) **mulsh6**
(sl(30:31) = 01)
au(24:31) = 0 **mulsh7**

rx — ad2 — a, s

au→ap→ao,au,pc di→b→alu irf→ird r1→ad→alu +2→au			irin
			2
			f502
asrrx1	cclrx1	h	bc04

pswh(20) = 0 **bctrz2**
pswh(20) = 1 **ashrx2**

dt→a→alu sh→b→alu			nacc
			2
			f506
mulsh5	mulsw5		mulsh4
dt→a→alu sh→b→alu			nacc
			1
			f509
mulsh6	mulsw6		mulsh4
alu→b→alu,at alu→a→alu,sfu rs16 0→sh→sl			nacc
			4
			f512
mulsh7	spmrr5		mulsh8
pc→ap→au sl→a→r1 +2→au			nacc
			0
			f514
mulsh8			ad1

rr — **ad1** — **fprrop**

agf0→a→ao,sl alu→dt edb→di	trro
	0
	3
	g002
fprr01 | | h	fprr02
ag06→a→sfu dile→b→alu,sh 0→alu ls4 sh←sl←0	nacc
	1
	3
	g004
fprr02 | |	fprr03
ag10→a→alu,au sfu→bsb sh→b→alu,au	nacc
	1s
	g006
fprr03 | |	bc14

alu = 0 **opexc1**
alu ≠ 0 **fprr04**

alu→dt ire→sfu	nacc
	0
	g009
fprr04 | |	fprr05
au→ao dol→edb sfu→b→do	swmc
	0
	g011
fprr05 | |	fprr06
alu→a→au au→ao dol→edb sfu→b→do 0→au	swmc
	0
	g013
fprr06 | |	ad2

rx — **ad1** — **fprxs**

dilz→*→au x2→*→au	nacc
	0
	3
	g102
fprx01 | |	fprx02
ag01→bd→alu au→bp→au rb→a→au 0→alu	nacc
	1
	1
	g104
fprx02 | |	fprx03
agf0→a→ao,sl alu→dt edb→di	trro
	0
	g106
fprx03 | fprr01 | h	fprx04
ag06→a→sfu dile→b→alu,sh 0→alu ls4 sh←sl←0	nacc
	1
	g108
fprx04 | fprr02 |	fprx05
au→*→at,au dt→*→au sfu→bsb	nacc
	0
	1
	g110
fprx05 | |	bc14

alu = 0 **opexc1**
alu ≠ 0 **fprx06**

au→ap→au di→ad→do dol→edb sh→b→ao,au	swmc
	0
	g114
fprx11 | |	ad2

ag10→a→au au→ao edb→di ire→sfu sh→b→au	trop
	0s
	1
	g201
fprx06 | | b	fprx07
at→bp→ao au→a→dt dt→bd→alu edb→di 0→alu	trop
	1
	g203
fprx07 | | h	fprx08
au→ao dol→edb pc→a→au sfu→b→do +2→au	swmc
	0
	g205
fprx08 | |	fprx09
au→pc dol→edb dt→a→ao,au sfu→b→do 0→au	swmc
	0
	g207
fprx09 | |	fprx12
di→a→do dol→edb sh→b→ao	swmc
	0
	g209
fprx12 | |	bc14

alu = 0 **fprx11**
alu ≠ 0 **fprx13**

dilz→*→au x2→*→au	nacc
	0
	g212
fprx11 | fprx01 |	fprx12
ag03→bd→alu au→bp→au rb→a→au 0→alu	nacc
	1
	1
	g214
fprx12 | |	fprx03

rx — **ad1** — **fprx1**

at→ap→au di→ad→do dol→edb sh→b→ao +2→au	swmc
	0
	g302
fprx13 | |	fprx10
ag10→a→au alu→bd→alu au→bp→ao,at edb→di −1→alu 0→au	trop
	1
	g304
fprx10 | | h	fprx12

rr — **ad2** — **fpcrop**

au→ao edb→dil	srmc
	0
	g307
fpcr07 | cdsr06 |	fpcr08
au→ao edb→dil	srmc
	0
	g309
fpcr08 | cdsr06 |	fpcr09
ag0c→b→alu di→a→alu,pur,sfu	nacc
	5
	1
	g311
fpcr09 | |	fpcr10
edb→irf pc→a→ao,au sfu→cc +2→au	trin
	0
	g313
fpcr10 | cead10 | h	bc14

alu = 0 **bcnop1**
alu ≠ 0 **fpcr11**

rrx — **ad1** — **fpexc**

agf0→a→ao,sl alu→dt edb→di	trro
	0
	g402
fpexc1 | fprr01 | h	fpexc2
ag06→a→sfu dile→b→alu,sh 0→alu ls4 sh←sl←0	nacc
	1
	g404
fpexc2 | fprr02 |	fpexc3
ag440000→bd→sh rs16 0→sh→sl	nacc
	0
	g406
fpexc3 | opex30 |	bc14

alu = 0 **opexc1**
alu ≠ 0 **spexc1**

edb→di,irf pc→a→ao,au +2→au	trin
	0
	g411
fprr10 | bcrrw2 | h	bc14

alu = 0 **bcnop1**
alu ≠ 0 **fpcr11**

ag0c→a→sl pswh→sfu pswl→b→alu −4→alu 0011101→etmssrr	nacc
	1
	g414
fpcr11 | |	opexc2

int — **a0** — **restart**

ag01→a→ao,at,au edb→di pswh→sfu +6→au	srak
	0
	g502
rstrt1 | |	rstrt6
ag86→bd→alu at→bp→ao au→a→au,sl edb→dil −2→alu +1→au 0011101→etmssrr	srak
	1
	g504
rstrt6 | exin11 |	rstrt7
alu→dt pswl→b→alu −4→alu ls8 sh←0	nacc
	1
	g506
rstrt7 | spexc2 |	rstrt2
au→ap→ao,at,au di→ad→sl doh→edb sfu→b→do +2→au	wrro
	0
	g508
rstrt2 | exin02 | w	rstrt3
au→ap→ao,au di→ad→sl dol→edb ilc→sfu sfu→b→do +2→au	wrro
	0
	g510
rstrt3 | exin03 | h	rstrt4
alu→*→do au→a→ao,au doh→edb +2→au	wrro
	0
	g512
rstrt4 | csrsw4 | w	rstrt5
ag0→a→au alu→*→do au→ao dol→edb 0→au	wrro
	0
	g514
rstrt5 | | h	exin07

rr ad1 clcl

au→pc r1(1)→bz→alu,at r2(1)→az→au −1→alu −1→au	nacc 1 h002
clcl01	bc17

r1 ≠ r3 clcl02
r1 = r3 clcl03

ex→b→alu pc→ap→au r2(1)→ad→sl −4→alu −4→au ls8 sh←sl←0	nacc 1 h005
clcl02 mvclr6	bc09

(alu < 0) (au < 0) clcl03
(alu ≥ 0) (au ≥ 0) clcl06
(alu ≥ 0) (au < 0) clcl05
(alu < 0) (au ≥ 0) clcl04

ag0→ad7→r1(1), sfu irf→ird pc→*p→ao r2→*→dt	irin 0 h008
clcl03 h	clcl34

alu→a→alu di→b→alu	nacc 2 h011
clcl16 clcl14	clcl17
r1(1)→bz→alu r2(1)→az→au 0→alu −1→au	nacc 1 h013
clcl17 clcl15	bc20

alub = 0 clcl21
alub ≠ 0 clcl22

alu→*→ex au→pc edb→dil r2→*→ao	trop 0 2 h101
clcl04 b	clcl12
r2→a→ao,au,dt sh→b→alu 0→alu +1→au	irop 1 h103
clcl12 b	clcl13
au→*→r2 edb→dil r1→*→at	frop 0 h105
clcl13	bc29

au(24:31) = 0 clcl16
au(24:31) ≠ 0 clcl14

alu→a→alu di→b→alu	nacc 2 4 h108
clcl14	clcl15
r1(1)→bz→alu r2(1)→az→au 0→alu −1→au	nacc 1 1 h110
clcl15	bc20

alub = 0 clcl31
alub ≠ 0 clcl22

at→bz→r1 dt→adz→r2 pc→ap→au +2→au	nacc 0 h113
clcl22	clcl36

alu→*→ex au→pc edb→dil r1→*→ao	trop 0 h201
clcl05 b	clcl08
r1→b→alu,at,ao +1→alu	irop 1 h203
clcl08 clcl07 b	clcl09
alu→*→r1 edb→di,irf r2→*→dt	frop 0 h205
clcl09	bc20

alub = 0 clcl23
alub ≠ 0 clcl10

di→a→alu sh→b→alu	nacc 2 1 h208
clcl10	clcl11
r1(1)→bz→alu r2(1)→az→au −1→alu 0→au	nacc 1 1 h210
clcl11	bc20

alub = 0 clcl32
alub ≠ 0 clcl22

alu→ad7→r1(1) au→b7→r2(1) pc→ap→au +2→au	nacc 0 h213
clcl21	bc07

(alu ≠ 0) + (au ≠ 0) start1
(alu = 0) (au = 0) clcl36

alu→*→ex au→pc edb→dil r2→*→ao	trop 0 h301
clcl06 clcl04 b	clcl07
r1→b→alu,at,ao +1→alu	irop 1 2 h303
clcl07 b	clcl25
dt→a→sfu edb→dil	frop 0 h305
clcl25	clcl26
alu→b→r1 r2→a→ao,au,dt +1→au	irop 0 h307
clcl26 b	bc20

alub = 0 clcl18
alub ≠ 0 clcl27

alu→ad7→r1(1) au→b7→r2(1)	nacc 0 h310
clcl30	bc14

alu = 0 clcl31
alu ≠ 0 clcl33

r1→b→alu,at,ao +1→alu	irop 1 h313
clcl33 clcl07 b	bc21

au = 0 clcl09
au ≠ 0 clcl25

au→a→au, r2 dibz→bd→alu edb→dil 0→alu −1→au	frop 1 1 h401
clcl27	bc29

au(24:31) = 0 clcl19
au(24:31) ≠ 0 clcl28

alu→a→alu di→b→alu	nacc 2 h404
clcl28 clcl14	clcl29
r1(1)→bz→alu r2(1)→az→au −1→alu −1→au	nacc 1 h406
clcl29 mvclb1	bc20

alub = 0 clcl30
alub ≠ 0 clcl22

di→a→alu sh→b→alu	nacc 2 h409
clcl23 clcl10	clcl24
r1(1)→bz→alu r2(1)→az→au −1→alu 0→au	nacc 1 h411
clcl24 clcl11	bc20

alub = 0 clcl21
alub ≠ 0 clcl22

au→bz→r1 pc→a→au +2→au	nacc 0 h414
clcl35	ad1

au→a→au,r2 dibz→bd→alu edb→dil 0→alu −1→au	frop 1 h501
clcl18 clcl27	clcl19
alu→a→alu di→b→alu	nacc 2 h503
clcl19 clcl14	clcl20
r1(1)→bz→alu r2(1)→az→au −1→alu −1→au	nacc 1 h505
clcl20 mvclb1	bc20

alub = 0 clcl21
alub ≠ 0 clcl22

alu→ad7→r1(1) au→b7→r2(1) pc→ap→au +2→au	nacc 0 h508
clcl31 clcl21	bc21

au = 0 clcl36
au ≠ 0 clcl12

alu→ad7→r1(1) au→b7→r2(1) pc→ap→au +2→au	nacc 0 h511
clcl32 clcl21	bc14

alu = 0 clcl36
alu ≠ 0 clcl08

at→*p→au dt→*dz→r2 edb→di,irf r1→*→au sfu→cc	frin 0 h514
clcl34	clcl35

rs
ad1
clm,icm
ag0→ad→dt
au→ap→ao,au
dilz→b→at
+2→au
irin
0
i002
icmr01 adssb1 h bc25
ire(12:13) = 0 icmr02
ire(12:13) ≠ 0 icmr08
at→bp→au
au→pc
dt→bd→sh
edb→irf
rb→a→au
frin
0
i005
icmr08 icmr02 icmr09
au→a→ao,au
edb→dil
+1→au
trop
0
i007
icmr09 icmr03 b bc27
ire(12:13) = 01 icmr10
ire(12:13) = 10 icmr12
ire(12:13) = 11 icmr11
au→a→ao,au
edb→dil
+1→au
ls8 sh←0
trop
0
1
i010
icmr17 b icmr06
au→a→ao,au
di→b1→sh1
edb→dil
+1→au
ls8 sh←0
trop
0
1
i012
icmr06 b icmr05
di→b1→sh1
nacc
0
2
i014
icmr05 ad2
au→a→ao,au
di→b1→sh1
edb→dil
+1→au
ls8 sh←0
trop
0
i101
icmr11 icmr06 b icmr10
di→b1→sh1
nacc
0
i103
icmr10 icmr05 bc26
ire(14:15) = 00 icmr13
ire(14:15) = 01 icmr14
ire(14:15) = 10 icmr16
ire(14:15) = 11 icmr17
di→b1→sh1
ls8 sh←0
nacc
0
i106
icmr12 icmr07 bc26
ire(14:15) = 00 icmr13
ire(14:15) = 01 icmr14
ire(14:15) = 10 icmr16
ire(14:15) = 11 icmr17
at→bp→au
au→pc
dt→bd→sh
edb→irf
rb→a→au
frin
0
1
i109
icmr02 icmr03
au→a→ao,au
edb→dil
+1→au
trop
0
1
i111
icmr03 b bc26
ire(14:15) = 00 icmr04
ire(14:15) = 01 icmr05
ire(14:15) = 10 icmr07
ire(14:15) = 11 icmr06
dt→b→alu
irf→ird
pc→ap→ao,au
0→alu
+2→au
irin
2
i114
icmr04 h bcnop2
rs
ad2
clm
irf→ird
pc→a→ao,au
r1→bm→dt
+2→au
irin
0
i202
icmr18 h icmr19
dt→a→alu
edb→di,irf
sh→b→alu
frin
2
i204
icmr19 ad1
alu→dt
pswl→b→alu
−4→alu
ls8 sh←0
nacc
1
i207
icmr22 spexc2 icmr23
sh→b→alu
0→alu
nacc
2
i209
icmr23 ad1
au→a→ao,au
edb→dil
+1→au
ls8 sh←0
trop
0
i212
icmr16 icmr17 b icmr07
di→b1→sh1
ls8 sh←0
nacc
0
1
i214
icmr07 ad2
rs
ad2
icm
edb→di,irf
irf→ird
pc→a→ao,au
sh→bm→r1
+2→au
trin
0
i302
icmr20 h bc27
ire(12:13) = 00 icmr21
ire(12:13) = 01 icmr22
ire(12:13) = 10 icmr23
ire(12:13) = 11 icmr23
alu→*→alu
dt→*→alu
ls16 sh←0
nacc
1
i305
icmr21 spkas5 bc26
ire(14:15) = 01 icmr22
ire(14:15) = 10 icmr23
ire(14:15) = 11 icmr23
au→a→ao,au
edb→dil
+1→au
ls16 sh←0
trop
0
i308
icmr14 b icmr05
alu→*→alu
dt→*→alu
ls16 sh←0
nacc
1
i311
icmr13 spkas5 ad2
rx
ad2
ic
au→a→ao,au,pc
di→b1→r1
irf→ird
+2→au
irin
0
i314
icrxb1 h bcnop2
rr
ad1
isk
ag0f→ad→alu,sfu
r2→b→alu,at
nacc
5
1
i402
iskrr1 iskrr2
r1→b→sh
sfu→bsb
nacc
0
3
i404
iskrr2 bc20
alub = 0 iskrr6
alub ≠ 0 spexc1
alu→dt
at→a→ao
edb→di
sfu→bd→alu
−1→alu
srmc
1
i407
iskrr6 mvos01 iskrr3
at→*→ao
di→*d→dt
edb→dil
srmc
0
i409
iskrr3 mvnss2 iskrr4
agfe→ad→alu
au→ap→ao,au,pc
di→b→alu
irf→ird
+2→au
irin
5
i411
iskrr4 h iskrr5
alu→b1→r1
edb→di,irf
frin
0
i413
iskrr5 ad1
s
ad1
ipk
agf0→ad→alu
au→ap→ao,au,pc
irf→ird
sh→b→alu
pswh→sfu
+2→au
irin
5
1
i502
ipks01 w ipks02
ag01→a→sfu
edb→di,irf
sfu→b→sh
rs16 0→sh→sl
frin
0
4
i504
ipks02 ipks03
agf0→ad→alu
au→ap→ao,au,pc
irf→ird
sh→b→alu
pswh→sfu
+2→au
irin
5
i506
ipks03 ipks01 h ipks04
alu→b1→re2
edb→di,irf
frin
0
i508
ipks04 ad1
rs
ad2
sld1,srd1
irf→ird
pc→ap→ao,au
r1→b→alu,sh
r1(1)→ad→alu,sl
+2→au
shift
irin
1
i512
slrds1 srdas1 h slrds2
edb→di,irf
sh→b→r1
sl→a→r1(1)
frin
0
i514
slrds2 ad1

rx	ad1 ↓	cvb
au→ao,pc dilz→b→au ×2→a→au		irin
		0
		j002
cvb01	adrxh1 h	cvb02
ag0→ad→dt au→ap→au edb→irf rb→b→au		frin
		0
		j004
cvb02	adrxh2	cvb03
ag0→a→au,sl au→bp→ao,at dt→bd→sh edb→di +7→au		trop
		0
		j006
cvb03	b	cvb04
at→ap→au au→b→alu,at di→ad→pur 0→alu +1→au		nacc
		1
		j008
cvb04		cvb05
purh→a→alu,sfu sh→b→alu		nacc
		1
		1
		j010
cvb05		bc24
aluc = 1		**cvb25**
aluc = 0		**cvb06**

alu→b→sh ls2 sl←sh←0	nacc
	0
	j013
cvb06	bc18
digit	**cvb07**
digit	**dtexc1**

alu→b→alu,dt at→a→au 0→alu +1→au	nacc
	4
	j101
cvb15	bc19
(pos) (alu ≥ 0)	**cvb16**
(pos) (alu < 0)	**cvb17**
neg	**cvb18**
digit	**dtexc1**

alu→bd→alu sh→b→alu	nacc
	1
	1
	j104
cvb07	bc24
aluc = 1	**cvb21**
aluc = 0	**cvb08**

alu→bd→sh ls1 sl→sh→0	nacc
	0
	j107
cvb08	bc24
aluc = 1	**cvb22**
aluc = 0	**cvb09**

at→ap→au au→bp→ao,at purl→ad→alu,sfu sh→bd→alu −1→au	irop
	1
	j110
cvb09 b	cvb10
alu→b→sh edb→di ls2 sl←sh←0	frop
	0
	j112
cvb10	bc18
digit	**cvb11**
digit	**dtexc1**

alu→a→alu sh→b→alu	nacc
	1
	j201
cvb11 cvb07	bc24
aluc = 1	**cvb23**
aluc = 0	**cvb12**

alu→bd→sh at→ap→au au→bp→at di→ad→pur +1→au ls1 sl←sh←0	nacc
	0
	j204
cvb12	bc21
au = 0	**cvb13**
au ≠ 0	**cvb05**

purh→a→alu,sfu sh→b→alu	nacc
	1
	j207
cvb13 cvb05	bc24
aluc = 1	**cvb24**
aluc = 0	**cvb14**

purl→a→sfu sh→b→dt	nacc
	0
	j210
cvb14	bc18
digit	**cvb19**
digit	**dtexc1**

alu→bd→dt,r1 at→bp→au sl→a→at,au,sfu	nacc
	0
	j213
cvb18	bc05
alu ≤ 0	**cvb17**
alu > 0	**ddr23**

ag01→a→sl	nacc
	0
	j301
cvb21 cvb20	cvb08
ag01→a→sl	nacc
	0
	j303
cvb22 cvb20	cvb09
ag01→a→sl	nacc
	0
	j305
cvb23 cvb20	cvb12
ag01→a→sl	nacc
	0
	j307
cvb24 cvb28	cvb14
ag01→a→sl	nacc
	0
	j309
cvb25 cvb20	cvb06
at→b→au sl→a→at,au,sfu	nacc
	0
	1
	j311
cvb16	cvb17
dt→*→r1 pc→a→au +2→au	nacc
	0
	j313
cvb17 csrsw3	bc21
au = 0	**bcnop1**
au ≠ 0	**ddr23**

rs	ad2 ↓	sla
irf→ird pc→ap→ao,au r1→b→alu,sh r1(1)→ad→alu,sl +2→au shift		irin
		1
		j402
slars1	srdas1 h	bc04
pswh(20) = 0		**slars2**
pswh(20) = 1		**slars5**

edb→di,irf sh→b→alu,r1 0→alu	frin
	2
	j405
slars2 srars2	bc08
ovf = 1	**slars3**
ovf = 0	**slars4**

edb→di,irf sh→b→alu,r1 0→alu	frin
	2
	j408
slars5 srars2	bc08
ovf = 1	**sldas4**
ovf = 0	**slars4**

wait	nacc
	0
	j411
cvb19 ddr11	bc24
aluc = 1	**cvb20**
aluc = 0	**cvb15**

ag01→a→sl	nacc
	0
	5
	j414
cvb20	cvb15

sfu→cc sh→b→r1	nacc
	0
	1
	j502
slars3	slars4
wait	nacc
	0
	j504
slars4 ddr11	ad1

rs	ad2 ↓	srda
irf→ird pc→ap→ao,au r1→b→alu,sh r1(1)→ad→alu,sl +2→au shift		irin
		1
		5
		j507
srdas1	h	srdas2
ag0→b→alu edb→di,irf sl→a→alu,r1(1)		frin
		2
		2
		j509
srdas2		srdas3
sh→b→alu,r1 0→alu		nacc
		3
		j511
srdas3		ad1

sfu→cc sh→b→r1	nacc
	0
	j514
sldas4 slars3	fpove1

rx	ad1	cvd
ag0c→ad→sfu au→ap→au r1→b→sh,alu 0→alu +2→au		nacc
		4
		k002
cvd01		cvd02
ag20→a→au au→pc 0→au		nacc
		0
		k004
cvd02		bc22
sh(0) = 0		**cvd04**
sh(0) = 1		**cvd24**

ag0c→a→sfu alu→b→sh	nacc
	0
	1
	k007
cvd24	cvd04
ag0→b→alu,at,dt au→a→au +1→alu −1→au ls1 sh←0	nacc
	1
	k009
cvd04	bc23
(sh(0) = 0)(au ≠ 0)	**cvd04**
(sh(0) = 1)(au ≠ 0)	**cvd05**
au = 0	**cvd15**

ag66666666→a →alu,sfu alu→dt dt→b→alu,at	nacc
	1
	k012
cvd12 cvd10	cvd13
alu→*→alu at→*→alu	nacc
	2
	k014
cvd11 cvd07	cvd16

ag66666666→* →alu alu→*→alu,at	nacc
	1
	k101
cvd05	bc23
(sh(0) = 0)(au ≠ 0)	**cvd06**
(sh(0) = 1)(au ≠ 0)	**cvd07**
au = 0	**cvd15**

alu→*→alu at→*→alu	nacc
	1
	1
	k104
cvd06	bc18
digit	**cvd09**
digit	**cvd08**

alu→b→alu au→ap→au corf→ad→alu −1→au ls1 sh←0	nacc
	2
	1
	k107
cvd08	bc24
aluc = 1	**cvd10**
aluc = 0	**cvd05**

ag66666666→a →alu,sfu alu→dt dt→b→alu,at	nacc
	1
	1
	k110
cvd10	cvd11

alu→b→alu corf→a→alu	nacc
	2
	k114
cvd16	cvd17

ag66666666→b →alu alu→dt dt→a→alu,at	nacc
	1
	k201
cvd17	bc23
(sh(0) = 0)(au ≠ 0)	**cvd06**
(sh(0) = 1)(au ≠ 0)	**cvd07**
au = 0	**cvd15**

alu→*→alu at→*→alu	nacc
	2
	1
	k204
cvd07	bc18
digit	**cvd09**
digit	**cvd08**

alu→b→alu au→ap→au corf→ad→alu −1→au ls1 sh←0	nacc
	2
	k207
cvd09 cvd08	bc24
aluc = 1	**cvd10**
aluc = 0	**cvd12**

at→a→sl	nacc
	0
	k210
cvd15a	cvd18
alu→*→alu at→*→alu	nacc
	1
	k212
cvd13 cvd06	cvd16
au→ap→ao,at,au di→ad→sl doh→edb sfu→b→do +2→au	wrro
	0
	k214
pint04 exin02 w	opexc3

dilz→b→au rb→a→au	nacc
	0
	k301
cvd15	cvd15a
au→bp→au r1→bd→sh x2→a→au	nacc
	0
	k303
cvd18	cvd19
au→a→at,au dt→b→sh +7→au ls4 sh←sl←0	nacc
	0
	k305
cvd19	bc22
sh(0) = 0	**cvd21**
sh(0) = 1	**cvd22**

ag0c→b→alu au→ao edb→di sl→ad→alu	trow
	2
	k308
cvd22 b	cvd14
ag0c→b→alu au→ao edb→di sl→ad→alu	trow
	1
	k310
cvd21 b	cvd14
at→a→ao,au doh→edb sh→b→do,dt +2→au	wrop
	0
	k312
cvd14 w	cvd20
au→a→ao,au dol→edb dt→*→do +2→au	wrop
	0
	15
	k314
cvd20 h	csrsw4

au→ao dol→edb r1→*→do	wrro
	0
	k401
eudu45 stcrx1 h	start0
sh→b→alu 0→alu msb→sfu	nacc
	1
	k403
eudu46 mulsw3	eudu38
au→ap→ao,au dol→edb dt→ad→do pswl→b→alu −4→alu +2→au inc r1	wrro
	1
	k405
eudu44 stmrs2 h	eudu43
au→a→ao,au doh→edb r1→b→do,dt +2→au	wrro
	0
	k407
eudu43 stmrs3 w	bc17
r1 ≠ r3	**eudu44**
r1 = r3	**eudu45**

int	a0	suppress program
ag20→a→ao,at,au edb→di pswh→sfu +7→au		srak
		0
		k410
pint01 prin01		pint02
ag8c→bd→alu at→bp→ao au→a→au,sl edb→dil 0→alu +1→au 0011101→etmssrr		srak
		1
		k412
pint02 prin08		pint03
alu→dt pswl→b→alu −4→alu ls8 sh←0		nacc
		1
		k414
pint03 spexc2		pint04

rr	ad1	ssk
ag0f→ad→alu,sfu r2→b→alu,at		nacc
		5
		k502
sskrr1 iskrr1		sskrr2
r1→b→sh sfu→bsb		nacc
		0
		k504
sskrr2 iskrr2		bc20
alub = 0		**sskrr3**
alub ≠ 0		**spexc1**

at→a→ao,au doh→edb sh→b→do,dt 0→au	swmc
	0
	k507
sskrr3	sskrr4
au→a→ao,au dol→edb dt→*→do +2→au	swmc
	0
	k509
sskrr4 cvd20	start1

s	ad1	ptlb
ag0c→a→sfu alu→b→sh		nacc
		0
		k512
ptlbs1 cvd24		ptlbs2
ag100→a→at,dt au→pc r1→b→sh sfu→bsb rs8 0→sh		nacc
		0
		k514
ptlbs2 stcm01		sskrr3

rx — **ad1** — **la**

au→ap→ao,au dilz→b→alu edb→irf x2→adz→alu +2→au	trin
	1
	l002
larxw1 | | w	larxw2
alu→*z→alu au→ap→ao,au,pc irf→ird rb→*z→alu +2→au	irin
	1
	l004
larxw2 | | h	larxw3
alu→*z→r1 edb→di,irf	frin
	0
	l006
larxw3 | |	ad1

rs — **ad2** — **lm**

alu→dt au→ao dilz→*→au edb→dil rb→*→au	trop
	0
	2
	l102
lmrsw1 | | b	lmrsw2
at→b→pc au→a→ao,au edb→di irf→ird +2→au	trop
	0
	l104
lmrsw2 | | w	lmrsw4
au→a→ao,au edb→dil r1→b1→dt +2→au rs32 0→sh→sl	trop
	0
	l106
lmrsw4 | cdsr04 | h	bc17

r1 ≠ r3 — **lmrsw5**
r1 = r3 — **lmrsw6**

rr — **ad1** — **lnr**

r2→b→alu,at 0→alu	nacc
	4
	1
	l202
lnrrr1 | |	lnrrr2
alu→dt au→a→ao,au,pc irf→ird r2→b→alu 0→alu +2→au	irin
	2
	l204
lnrrr2 | ltrrr1 | h	bc02

alu < 0 — **lnrrr3**
alu ≥ 0 — **bctrz2**

dt→b→alu,r1 edb→di,irf 0→alu	frin
	2
	l207
lnrrr3 | |	ad1

rx — **ad2** — **lh**

au→a→ao,au,pc dile→b→r1 irf→ird +2→au	irin
	0
	l009
lhrxh1 | | h	bcnop2
au→ap→ao,au di→ad→r1 edb→di +2→au inc r1	trop
	0
	l109
lmrsw5 | | w	lmrsw4
di→*→r1 edb→di,irf pc→a→ao,au +2→au	trin
	0
	l111
lmrsw6 | | h	ad1

rx — **ad2** — **l**

au→a→ao,au,pc di→b→alu irf→ird 0→alu +2→au	irin
	1
	l210
llrrx1 | bcnop1 | h	bctrz2

rr — **ad1** — **ltr**

alu→dt au→a→ao,au,pc irf→ird r2→b→alu 0→alu +2→au	irin
	2
	4
	l114
ltrrr1 | | h	bctrz2

rr — **ad1** — **lcr**

alu→dt au→a→ao,au,pc irf→ird r2→b→alu 0→alu +2→au	irin
	2
	l213
lcrrr1 | ltrrr1 | h	bc04

pswh(20) = 0 — **bctrz2**
pswh(20) = 1 — **ashrx2**

rr — **ad1** — **lpr**

r2→b→alu,at 0→alu	nacc
	4
	l302
lprrr1 | lnrrr1 |	bc04

pswh(20) = 0 — **lprrr2**
pswh(20) = 1 — **lprrr4**

alu→dt au→a→ao,au,pc irf→ird r2→b→alu 0→alu +2→au	lrin
	2
	l305
lprrr2 | ltrrr1 | h	bc02

alu ≥ 0 — **bctrz2**
alu < 0 — **lprrr5**

alu→dt au→a→ao,au,pc irf→ird r2→b→alu 0→alu +2→au	irin
	2
	l308
lprrr4 | ltrr1 | h	bc02

alu ≥ 0 — **bctrz2**
alu < 0 — **lprrr5**

alu→b→alu at→a→r1 edb→di,irf 0→alu	frin
	2
	l311
lprrr5 | lprrr3 |	bc03

aluv = 0 — **slars4**
aluv = 1 — **fpovel1**

alu→b→alu at→a→r1 edb→di,irf 0→alu	frin
	2
	1
	l314
lprrr3 | |	ad1

rr — **ad1** — **spm**

at→ap→au pswh→sfu r1→b→alu,au,sh r2→ad→alu rs16 0→sh→sl	nacc
	4s
	l402
spmrr1 | |	spmrr2
agffff8000→a→alu sfu→b→alu	nacc
	5
	1
	l404
spmrr2 | |	spmrr3
ag3f00→a→alu alu→dt sh→b→alu	nacc
	5
	1
	l406
spmrr3 | |	spmrr4
alu→*→alu dt→*→alu ls16 sh←0	nacc
	1
	l408
spmrr4 | spkas5 |	spmrr5
alu→b→alu,at alu→a→alu,sfu rs16 0→sh→sl	nacc
	4
	1
	l410
spmrr5 | |	ssms06
at→a→ao,au doh→edb pc→b→do,dt 0→au	swak
	0
	l412
spmrr6 | |	lpsw14

si — **ad1** — **stnsm,stosm**

au→pc dilz→a→at,au pswh→sfu sfu→b→alu,dt 0→alu 0→au	nacc
	1
	l502
stosm1 | ssms01 |	stosm2
at→bp→au rb→a→au sfu→bd→alu,sh 0→alu rs16 0→sh→sl	nacc
	1
	l504
stosm2 | ssms02 |	stosm3
alu→a→alu,sl pswh→sfu sfu→b→alu ls8 sh←sl←0	nacc
	1
	l506
stosm3 | |	stosm4
au→ao dol→edb dt→a→alu sh→b→alu,do	wrop
	2
	l508
stosm4 | | b	stosm5
agb8→a→alu alu→b→alu,sh rs8 0→sh→sl	nacc
	5
	l510
stosm5 | |	ssms05

rs — **ad1** — **store mim diag.**

ag07→b→alu au→a→alu,sl mim→sfu	nacc
	5
	l514
qmim01 | csrsw1 |	mvisi2

ssbm — ad1 — addr

au→a→ao,au l1→sfu sfu→b→alu,sh 0→alu +2→au	irin
	1
	2
	m002
amssb1 | | h	amssb2
alu→dt au→pc dilz→b→au edb→di,irf rb→a→au	frin
	0
	m004
amssb2 | bxhrx2 |	amssb3
au→ap→ao,au dt→ad→alu sfu→b→alu,au	irop
	4
	m006
amssb3 | | b	apssb5

si — ad1 — mvi

au→b→alu 0→alu	nacc
	4
	m010
mvisi1 | ddr20 |	mvisi2
au→pc dilz→*→au rb→*→au	nacc
	0
	6
	m012
mvisi2 | |	mvisi3
ag8c→ad→dt au→ap→ao,au dol→edb ilc→sfu sfu→b→do +2→au	wrop
	0
	9
	m014
mvisi3 | | b	start1

s — ad1 — spka

au→pc dilz→a→at,au pswh→sfu sfu→b→alu,dt 0→alu 0→au	nacc
	1
	m102
spkas1 | ssms01 |	spkas2
at→bp→au rb→a→au sfu→bd→alu,sh 0→alu rs16 0→sh→sl	nacc
	1
	m104
spkas2 | ssms02 |	spkas3
agf0→a→alu au→b→alu bsb→sfu	nacc
	5
	1
	m106
spkas3 | |	spkas4
agff0f→a→alu alu→dt sh→b→alu	nacc
	5
	m108
spkas4 | |	spkas5
alu→*→alu dt→*→alu ls16 sh←0	nacc
	1
	3
	m110
spkas5 | |	spkas6
alu→b→alu,sh irf→ird 0→alu rs16 0→sh→sl	nacc
	5
	1
	m112
spkas6 | |	spkas7
au→ap→au sh→b→au sl→ad→sfu	nacc
	0
	m114
spkas7 | alsm05 |	ssms05

ss — ad2 — mvz

alu→bd→alu at→a→ao,au au→bp→at edb→di −1→alu +1→au	trop
	1
	m202
mvzss1 | mvnss1 | b	mvzss2
at→*→ao di→*d→dt edb→dil	trow
	0
	m204
mvzss2 | mvnss2 | b	mvzss3
di→b→pur dt→a→pur sfu→ird	nacc
	0
	1
	m206
mvzss3 | |	mvzss4
at→a→ao,au au→bp→at dol→edb pur→bd→do +1→au	wrop
	0
	m208
mvzss4 | mvnss4 | b	bc02
alu<0	**start1**
alu≥0	**mvzss1**

ss — ad2 — mvc

at→a→ao,au au→b→at di→*d→dt edb→dil +1→au	trop
	0
	1
	m211
mvcss1 | | b	mvcss2
alu→bd→alu at→ap→ao,au au→bp→at di→ad→do dol→edb −1→alu +1→au	wrop
	1
	2
	m213
mvcss2 | | b	bc14
alu=0	**start1**
alu≠0	**mvcss1**

ss — ad2 — mvn

alu→bd→alu at→a→ao,au au→bp→at edb→di −1→alu +1→au	trop
	1
	1
	m302
mvnss1 | | b	mvnss2
at→*→ao di→*d→dt edb→dil	trow
	0
	4
	m304
mvnss2 | | b	mvnss3
di→a→pur dt→b→pur pswh→sfu	nacc
	0
	1
	m306
mvnss3 | |	mvnss4
at→a→ao,au au→bp→at dol→edb pur→bd→do +1→au	wrop
	0
	1
	m308
mvnss4 | | b	bc02
alu<0	**start1**
alu≥0	**mvnss1**

ag0→a→pur,sfu dt→b→alu −1→alu	nacc
	1
	1
	m311
mvos08 | |	mvos09
alu→dt au→ap→ao,au dol→edb pur→ad→do sfu→b→pur −1→au	wrop
	0
	m313
mvos09 | | b	bc02
alu<0	**start1**
alu≥0	**mvos08**

ss — ad2 — mvo

alu→dt at→a→ao edb→di sfu→bd→alu −1→alu	trop
	1
	2
	m402
mvos01 | | b	mvos05
au→a→ao,au di→b→pur edb→di −1→au	trop
	0
	m404
mvos02 | pack12 | b	mvos05
alu→dt di→a→pur,sfut dt→b→alu −1→alu	nacc
	1
	m406
mvos05 | pack13 |	bc02
alu<0	**mvos03**
alu≥0	**mvos06**

at→ap→ao,au au→bp→at dol→edb pur→ad→do sfu→bd→pur −1→au	wrop
	0
	2
	m409
mvos06 | | b	bc02
alu<0	**mvos08**
alu≥0	**mvos07**

alu→dt at→a→ao,au au→bp→at dt→bd→alu edb→di −1→alu −1→au	trop
	1
	m412
mvos07 | unpk09 | b	mvos05
at→ap→ao,au au→bp→at dol→edb pur→ad→do sfu→bd→pur −1→au	wrop
	0
	m414
mvos03 | mvos06 | b	start1

ss — ad2 — mvcin

at→a→ao,au au→b→at edb→dil −1→au	trop
	0
	m502
mvcin1 | | b	mvcin2
alu→bd→alu at→ap→ao,au au→bp→at di→ad→do dol→edb −1→alu +1→au	wrop
	1
	m504
mvcin2 | mvcss2 | b	bc14
alu=0	**start1**
alu≠0	**mvcin1**

s — ad1 — stidp

ag3700400→b→at,dt au→a→au +2→au	nacc
	0
	m507
stidp1 | |	stidp2
au→pc dilz→*→au rb→*→au	nacc
	0
	m509
stidp2 | mvisl2 |	stidp3
ag01→a→sfu au→b→alu +7→alu	nacc
	5
	1
	m511
stidp3 | |	stidp4
alu→a→ao dt→b→alu edb→dil 0→alu	trow
	1
	m513
stidp4 | | b	bc14
alu=0	**spsw05**
alu≠0	**spexc1**

rr — ad1 — mvcl

au→pc r1(1)→bz→alu r1(1)→bz→at,au r2(1)→az→au 0→alu	nacc
	1s
	n002
mvclr1	mvclr2
at→bp→au r2→az→at,au r2(1)→bdz→alu 0→alu	nacc
	1
	n004
mvclr2	bc15

(alu = 0) (au ≥ 0) mvcl19
(alu ≠ 0) (au ≥ 0) mvclr3
(alu = 0) (au < 0) mvcl19
(alu ≠ 0) (au < 0) mvclr4

r2→*z→au r2(1)→*z→au	nacc
	0
	n007
mvclr3	bc14

alu = 0 mvcl18
alu ≠ 0 mvclr4

agff000000→*→alu au→*→alu,dt	nacc
	5
	n010
mvclr4	mvclr2
at→bp→au dt→bdz→alu pswh→sfu r1→az→alu,au,dt	nacc
	4s
	n012
mvclr5	bc14

alu = 0 mvclr7
alu ≠ 0 mvclr6

ex→b→alu pc→ap→au r2(1)→ad→sl −4→alu −4→au ls8 sh←sl←0	nacc
	1
	3
	n101
mvclr6	bc12

(alu ≥ 0) (au ≥ 0) mvclr9
(alu < 0) + (au < 0) mvclr8

ex→b→alu pc→ap→au r2(1)→ad→sl −4→alu −4→au ls8 sh←sl←0	nacc
	1
	n104
mvclr7 mvclr6	bc01

(alu ≥ 0) + (au ≥ 0) mvclr9
(alu < 0) (au < 0) mvclr8

alu→*→ex au→pc edb→dil r2→*→ao	trop
	0
	n107
mvclr9 clc104 b	mvcl16
alu→*→ex au→pc r1→*→ao	irop
	0
	n109
mvcl16 b	mvcl17
edb→dil r1(1)→az→alu r2(1)→bz→alu	frop
	2
	n111
mvcl17	mvclb1

ex→b→alu pc→ap→au r2(1)→ad→sl −4→alu −4→au ls8 sh←sl←0	nacc
	1
	n114
mvcl18 mvclr6	mvcl16

ag03→ad→sfu at→bz→r2 irf→ird pc→ap→ao,au +2→au	irin
	0
	n201
mvclr8 h	mvcl20
dt→*z→r1 edb→di,irf sfu→cc	frin
	0
	n203
mvcl20	slars4

alu→ad7→r1(1) at→ap→ao,au dol→edb sh→b→do +1→au	wrop
	0
	n206
mvclb8 mvclb3 b	mvcl12

irf→ird pc→ap→ao,au r1→adz→dt r2(1)→bz→alu 0→alu +2→au	irin
	2
	n209
mvcl19 h	mvcl21
at→az→r2 dt→bz→r1 edb→di,irf	frin
	0
	n211
mvcl21	ad1

at→*z→r1 edb→di,irf	frin
	0
	n214
mvcl22	ad1

r1(1)→bz→alu r2(1)→az→au −1→alu −1→au	nacc
	1
	2
	n301
mvclb1	mvclb2
r1→*→at r2→*z→dt	nacc
	0
	n303
mvclb2	bc15

(alu = 0) (au ≥ 0) mvcl10
(alu ≠ 0) (au ≥ 0) mvclb5
(alu = 0) (au < 0) mvclb8
(alu ≠ 0) (au < 0) mvclb3

au→b7→r2(1) edb→dil r2→a→ao,au +1→au	trop
	0
	n306
mvcl10 b	mvcl11
at→ap→ao,au au→b→r2,dt di→ad→do dol→edb +1→au	wrop
	0
	n308
mvcl11 b	mvcl12
alu→ad7→r1(1) au→b→at pc→ap→au +2→au	nacc
	0
	n310
mvcl12	mvcl13
au→a→ao,au edb→irf ex→b→alu +4→alu +2→au	trin
	1
	1
	n312
mvcl13 w	mvcl14
alu→b→ex au→ap→ao,au,pc dt→adz→r2 irf→ird +2→au	irin
	0
	n314
mvcl14 h	mvcl22

alu→ad7→r1(1) at→ap→ao,au dol→edb sh→b→do +1→au	wrop
	0
	1
	n401
mvclb3 b	mvclb4
au→*→r1	nacc
	0
	n403
mvclb4	bc29

au(24:31) = 0 start1
au(24:31) ≠ 0 mvclb1

au→b7→r2(1) r2→a→ao,au +1→au	irop
	0
	n406
mvclb5 b	mvclb6
alu→ad7→r1(1) at→ap→au au→b→r2 edb→dil +1→au	frop
	0
	n408
mvclb6	bc29

au(24:31) = 0 mvcl15
au(24:31) ≠ 0 mvclb7

at→*p→ao au→*→r1 di→*d→do dol→edb	wrop
	0
	1
	n411
mvclb7 b	bc29

au(24:31) = 0 start1
au(24:31) ≠ 0 mvclb1

at→*p→ao au→*→r1 di→*d→do dol→edb	wrop
	0
	n414
mvcl15 mvclb7 b	start1

s — ad1 — rrb

ag0→*→at,au au→pc dilz→*→au	nacc
	0
	n502
rrbs01	rrbs02
ag0f→ad→sfu au→ap→au rb→b→au	nacc
	0
	n504
rrbs02 basxw2	rrbs03
au→be→at sfu→bsb	nacc
	0
	n506
rrbs03	rrbs04
alu→dt at→a→ao edb→di sfu→bd→alu −1→alu	srmc
	1
	n508
rrbs04 mvos01	rrbs05
ag01→b→alu at→*p→ao di→a→dt edb→dil −6→alu	srmc
	1
	n510
rrbs05	rrbs06
alu→b→alu at→*p→ao di→a→alu,do,sl doh→edb rs1 0→sh→sl	swmc
	5
	n512
rrbs06	rrbs07
alu→b→do at→*p→ao dol→edb sl→a→sfu	swmc
	0
	n514
rrbs07 noxsi2	cead10

rs — ad1 ↓ — lpsw diag.

age8→a→sfu dilz→b→at			nacc
			0
			o002
dpsw01			dpsw02
ag30→bd→alu at→bp→au rb→a→au sfu→ire −6→alu			nacc
			1
			o004
dpsw02			dpsw03
alu→a→sfu,au au→b→alu,at +7→alu +7→au xxxxx00→etmssrr			nacc
			5
			o006
dpsw03			dpsw04
au→ap→au r1→b→sh r1(1)→ad→sl +7→au			nacc
			0
			2
			o008
dpsw04			bc20

alub = 0 — **dpsw05**
alub ≠ 0 — **spexc1**

au→bp→at at→ap→ao,au sh→bd→r1 sl→ad→r1(1) +2→au			irop
			0
			o011
dpsw05		w	dpsw06
edb→di r2→b→sh r2(1)→a→sl			frop
			0
			o013
dpsw06			dpsw07

au→ap→ao,au sfu→ire sh→b→r2 sl→a→r2(1) +2→au			irop
			0
			o101
dpsw07		h	dpsw08
edb→dil r1→b→sh r1(1)→a→sl			frop
			0
			o103
dpsw08			dpsw09
agb80840ff→b →alu au→ap→ao,au di→ad→alu,dt,sfu +2→au			irop
			5
			o105
dpsw09		w	dpsw10
edb→di sh→b→r1 sl→a→r1(1)			frop
			0
			o107
dpsw10			dpsw11
ag80000→ad→alu alu→b→alu at→ap→au au→ao 0→au			irop
			6
			o109
dpsw11		h	dpsw12
edb→dil r2→b→sh r2(1)→a→sl			frop
			0
			o111
dpsw12			dpsw14
ird→ire sfu→pswh sh→b→r2 sl→a→r2(1)			nacc
			0
			o113
dpsw14			dpsw15

au→ap→au ilc→sfu r1→b→sh r1(1)→ad→sl +7→au			nacc
			0
			o201
dpsw15			dpsw16
sfu→ire sh→b→r1 sl→a→r1(1)			nacc
			0
			2
			o203
dpsw16			dpsw17
au→ap→au r1→b→sh r1(1)→ad→sl +7→au			nacc
			0
			o205
dpsw17	dpsw04		dpsw18
au→a→sfu			nacc
			0
			o207
dpsw18			dpsw19
sfu→ire sh→b→r1 sl→a→r1(1)			nacc
			0
			o209
dpsw19	dpsw16		dpsw20
r2→b→sh r2(1)→a→sl			nacc
			0
			o211
dpsw20			dpsw21
sh→b→r2 sl→a→r2(1)			nacc
			0
			o213
dpsw21			dpsw22

au→ap→au r1→b→sh r1(1)→ad→sl +7→au			nacc
			0
			o301
dpsw22	dpsw04		dpsw23
sfu→ire sh→b→r1 sl→a→r1(1)			nacc
			0
			o303
dpsw23	dpsw16		bc14

alu = 0 — **dpsw13**
alu ≠ 0 — **lpsw17**

di→b→do,dt,pc dt→a→ao,au doh→edb 0→au 0x00011→etmssrr			swak
			0
			o306
dpsw13			lpsw14

at→ap→au au→ao di→ad→sl dt→b→at edb→dil 0→au ls8 sh←sl←0			trop
			0
			o309
mvcw13	mvcw12	h	bc26

ire(14:15) = 00 — **stcm09**
ire(14:15) = 01 — **mvcw15**
ire(14:15) = 10 — **mvcw16**
ire(14:15) = 11 — **mvcw10**

au→a→ao,au di→*→do,sh doh→edb +2→au rs8 0→sh			wrop
			0
			o312
mvcw16	mvcw15	h	stcm09
au→a→ao,au di→*→do,sh doh→edb +2→au rs8 0→sh			wrop
			0
			1
			o314
mvcw15		h	start1

ss — ad1 ↓ — mvc(wl)

au→ap→ao,au dilz→ad→alu sfu→b→alu,at,dt +2→au			irin
			1
			o402
mvcw01	alssb1	h	mvcw02
alu→*→au au→pc edb→di,irf mim→sfu rb→*→au			frin
			0
			o404
mvcw02	alssb2		mvcw03
at→a→au au→bp→ao,au dilz→bd→alu edb→dil 0→alu			trow
			1s
			o406
mvcw03		b	mvcw04
alu→*→alu au→*p→at rb→*→alu			nacc
			1
			o408
mvcw04	alssb4		mvcw05
alu→a→alu,au,sl dt→b→alu 0→au			nacc
			1
			o410
mvcw05			mvcw06
alu→b→ao at→a→sl			irop
			0
			o412
mvcw06		b	bc28

sl(30:31) = 0 — **mvcw07**
sl(30:31) ≠ 0 — **alssb6**

ag0→ad→sfu at→ap→au au→bp→at dt→bd→alu edb→dil 0→alu 0→au			frop
			1
			o501
mvcw07	alssb6		bc28

sl(30:31) = 0 — **mvcw08**
sl(30:31) ≠ 0 — **mvcss1**

alu→bd→alu at→a→ao,au au→bp→at −4→alu +4→au			irop
			1
			o504
mvcw08		w	mvcw09
au→a→au,dt edb→di −2→au			frop
			0
			o506
mvcw09			bc02

alu < 0 — **mvcw13**
alu ≥ 0 — **mvcw12**

at→ap→au au→ao di→ad→sl dt→b→at edb→dil 0→au ls8 sh←sl←0			trop
			0
			1
			o509
mvcw12		h	mvcw10
au→a→ao,au di→*→do,dt doh→edb +2→au			wrop
			0
			o511
mvcw10	eudu05	w	mvcw11
au→a→ao,au dol→edb dt→*→do +2→au			wrop
			0
			o513
mvcw11	cvd20	h	bc02

alu < 0 — **start1**
alu ≥ 0 — **mvcw08**

ss	ad2	pack
alu→b→alu au→a→ao,au edb→di −1→alu −1→au		trop 1 1 p002
pack01	b	pack02
alu→dt di→a→sfut sfu→b→alu −1→alu		nacc 1 1 p004
pack02		bc06
alu > 0		**pack10**
alu = 0		**pack03**
alu < 0		**pack04**

alu→dt at→a→ao,au au→bp→at dol→edb sfu→bd→alu,do 0→alu −1→au	wrop 5 p007
pack04 pack03 b	bc06
alu < 0	**start1**
alu = 0	**pack18**
alu > 0	**pack06**

alu→dt dt→b→alu −1→alu	nacc 1 1 p010
pack06	pack07
alu→b→alu au→ap→ao,au dol→edb dt→ad→do −1→alu −1→au	wrop 1 p012
pack07 b	bc02
alu < 0	**start1**
alu ≥ 0	**pack07**

alu→dt at→a→ao,au au→bp→at dol→edb sfu→bd→alu,do 0→alu −1→au	wrop 5 1 p101
pack03 b	bc02
alu < 0	**start1**
alu ≥ 0	**pack08**

alu→dt at→a→ao,au au→bp→at dt→bd→alu edb→di −1→alu −1→au	trop 1 p104
pack08 unpk09 b	pack17
au→a→au di→b→pur +1→au	nacc 0 p106
pack17	pack09
alu→dt at→a→ao,au au→bp→at dol→edb pur→bd→alu,do 0→alu −1→au	wrop 5 p108
pack09 pack15 b	bc06
alu < 0	**start1**
alu = 0	**pack18**
alu > 0	**pack06**

alu→dt at→a→ao,au au→bp→at dol→edb pur→bd→alu,do 0→alu −1→au	wrop 5 p111
pack16 pack15 b	bc06
alu < 0	**start1**
alu = 0	**pack18**
alu > 0	**pack06**

alu→*→do au→a→ao,au dol→edb +2→au	wrop 0 3 p114
pack18 b	start1

ag0→ad→pur at→ap→ao,au au→bp→at dol→edb sfu→bd→do −1→au	wrop 0 2 p201
pack10 b	bc02
alu < 0	**start1**
alu ≥ 0	**pack11**

alu→dt at→a→ao,au au→bp→at dt→bd→alu edb→di −2→alu −1→au	trop 1 p204
pack11 b	pack12
au→a→ao,au di→b→pur edb→di −1→au	trop 0 1 p206
pack12 b	pack13
alu→dt di→a→pur,sfut dt→b→alu −1→alu	nacc 1 1 p208
pack13	bc06
alu > 0	**pack14**
alu = 0	**pack15**
alu < 0	**pack16**

alu→dt at→a→ao,au au→bp→at dol→edb pur→bd→alu,do 0→alu −1→au	wrop 5 2 p211
pack15 b	bc02
alu < 0	**start1**
alu ≥ 0	**pack08**

alu→a→ao,au edb→di sh→b→alu,au,dt 0→alu	trop 1 1 p214
apssb8 b	ad2

ssbp	ad1	addr
au→a→ao,au l1→sfu sfu→b→alu,sh 0→alu +2→au		irin 1 p302
apssb1 amssb1 h		apssb2
alu→dt au→pc dilz→b→au edb→di,irf rb→a→au		frin 0 p304
apssb2 bxhrx2		apssb3
dt→a→alu sfu→b→alu,at		nacc 4 p306
apssb3		apssb4
alu→ad→alu at→*p→au au→*p→ao,au sfu→bd→alu		irop 4 p308
apssb4 b		apssb5
alu→dt au→ap→at,au dilz→b→alu edb→dil rb→a→alu +2→au		frop 1 p310
apssb5 adssb2		bc05
alu > 0		**apssb6**
alu ≤ 0		**apssb8**

at→ap→ao,au au→bp→at dol→edb pur→ad→do sfu→bd→pur −1→au	wrop 0 1 p313
pack14 mvos06 b	bc02
alu < 0	**start1**
alu ≥ 0	**pack11**

rs	ad1	sbcd diag.
ag0→ad→dt au→ap→ao,au dilz→b→at +2→au		irin 0 p402
sbcd01 adssb1 h		abcd02
au→ap→ao,au edb→irf ex→b→alu r1→adz→sl +4→alu +2→au		trin 1 p405
clcl36 w		clcl37
alu→b→ex au→ap→ao,au,pc irf→ird r2→adz→dt +2→au		irin 0 p407
clcl37 h		clcl38
dt→bz→r2 edb→di,irf sl→az→r1		frin 0 p409
clcl38		ad1
alu→*→au dt→*→au		nacc 0 1 p412
apssb6		apssb7
alu→a→au au→ao edb→dil sh→b→alu,au 0→alu		trop 1 p414
apssb7 b		ad2

rs	ad1	abcd diag.
ag66666666→ad →dt au→ap→ao,au dilz→b→at +2→au		irin 0 p502
abcd01 h		abcd02
au→pc dilz→*→alu,au edb→di,irf rb→*→alu,au		frin 1 p504
abcd02 adrsh2		abcd03
au→ap→au→ao dt→*→alu edb→di r2→b→alu +2→au		trop 1 p506
abcd03 w		abcd07
au→ao edb→dil		trop 0 p508
abcd07 cdsr06 h		abcd04
alu→a→alu di→b→alu		nacc 2 p510
abcd04 clcl14		abcd05
alu→b→alu corf→ad→alu irf→ird pc→ap→ao,au +2→au		irin 3 p512
abcd05 h		abcd06
alu→b→r2 edb→di,irf		frin 0 p514
abcd06		ad1

int	a0	machine check w/ external
ag30→a→ao,at,sl edb→di exb→sfu 0011101→etmssrr		srak 0 1 q002
mcin01		mcin02
ag08→ad→sfu at→ap→ao,au edb→dil pswl→b→alu −4→alu 0→au		srak 1 1 q004
mcin02		mcin12
r1→b→sh sfu→bsb		nacc 0 q006
mcin12	iskrr2	mcin03
au→a→ao,at,au di→b→dt,sh edb→di pswh→sfu +2→au		srmc 0 q008
mcin03	adrxw3	mcin04
at→*→ao di→*d→dt edb→dil		srmc 0 q010
mcin04	mvnss2	mcin05
ag88→ad→dt at→ap→ao,au doh→edb sfu→b→do +2→au		wrro 0 q012
mcin05	svcrr2 | w	mcin06
au→ap→ao,au di→ad→sl dol→edb ilc→sfu sfu→b→do +2→au		wrro 0 q014
mcin06	exin03 | h	mcin07

alu→*→do au→a→ao,au doh→edb +2→au	wrro 0 q101
mcin07 | csrsw4 | w	mcin08
ag40→*→au alu→*d→do at→*p→au au→ao dol→edb	wrro 0 q103
mcin08 | exin05 | h	mcin09
age8→a→ao,au au→bp→at doh→edb sh→bd→do,dt +2→au	wrro 0 q105
mcin09 | w	mcin10
au→a→ao,au dol→edb dt→*→do +2→au	wrro 0 q107
mcin10 | cvd20 | h	mcin11
au→ap→ao,au doh→edb mim→sfu sl→ad→do,dt +2→au	wrro 0 q109
mcin11 | eudu13 | w	prin07

rs	ad2	sll,srl
irf→ird pc→ap→ao,au r1→b→alu,sh r1(1)→ad→alu,sl +2→au shift		irin 1 q112
slrrs1 | srdas1 | h		slrrs2
edb→di,irf sh→b→r1		frin 0 2 q114
slrrs2		ad1

s	ad1	lpsw
x1xxxxx→etmssrr		nacc 0 q202
lpsw01 | trace1		lpsw02
au→pc dilz→*→au rb→*→au		nacc 0 q204
lpsw02 | mvisi2		lpsw03
ag01→a→sfu au→b→alu +7→alu		nacc 5 q206
lpsw03 | stidp3		lpsw04
au→a→pc,sl xxxxx11→etmssrr		nacc 0 q208
lpsw04 | srsdr3		bc14
alu = 0 **alu ≠ 0**		**lpsw06** **spexc1**

di→b→do,dt,pc dt→a→ao,au,sl doh→edb 0→au 0011101→etmssrr	swak 0 q211
lpsw13	lpsw14
au→a→ao,au dol→edb dt→*→do +2→au	swak 0 q213
lpsw14 | cvd20	bc10
pswh(14) = 0 **pswh(14) = 1**	**start1** **wait01**

au→a→ao,au di→b→dt,sh edb→di +2→au	trop 0 q301
lpsw06 | cdsr03 | w	lpsw07
au→a→ao,au edb→dil r1→b1→dt +2→au rs32 0→sh→sl	trop 0 q303
lpsw07 | cdsr04 | h	lpsw08
agb80840ff→b →alu au→ap→ao,au di→ad→alu,dt,sfu edb→di +2→au	trop 5 2 q305
lpsw08 | w	lpsw09
ag80000→a→alu alu→b→alu au→ao edb→dil	trop 6 q307
lpsw09 | exin10 | h	lpsw10
sfu→pswh	nacc 0 q309
lpsw10 | ssms06	bc14
alu = 0 **alu ≠ 0**	**lpsw13** **lpsw17**

ag0→a→pur,sfu dt→b→alu −1→alu	nacc 1 q312
lpsw17 | mvos08	lpsw11
ag06→a→sl di→b→alu sfu→ilc 0→alu 0011101→etmssrr	nacc 1 q314
lpsw11	lpsw16

doh→edb dt→b→alu,ao sl→a→do +2→alu	wrro 1 q401
exin06 | w	exin13
alu→b→ao dol→edb sl→a→do	wrro 0 q403
exin13 | h	exin07
au→a→ao,au di→b→dt,sh edb→di +2→au	trro 0 q405
exin07 | cdsr03 | w	exin08
au→a→ao,au edb→dil r1→b1→dt +2→au rs32 0→sh→sl	trro 0 q407
exin08 | cdsr04 | h	exin09
agb80840ff→b →alu au→ap→ao,au di→ad→alu,dt,sfu edb→di +2→au	trro 5 q409
exin09 | lpsw08 | w	exin10
ag80000→a→alu alu→b→alu au→ao edb→dil	trro 6 2 q411
exin10 | h	lpsw10

di→a→pur dt→b→pur pswh→sfu	nacc 0 q414
lpsw16 | mvnss3	opexc2

int	a0	external
ag10→a→ao,at,au edb→di pswh→sfu +7→au		srak 0 q502
exin01		exin11
ag86→bd→alu at→bp→ao au→a→au,sl edb→dil −2→alu +1→au 0011101→etmssrr		srak 1 1 q504
exin11		exin12
alu→dt pswl→b→alu −4→alu ls8 sh←0		nacc 1 q506
exin12 | spexc2		exin02
au→ap→ao,at,au di→ad→sl doh→edb sfu→b→do +2→au		wrro 0 3 q508
exin02	w	exin03
au→ap→ao,au di→ad→sl dol→edb ilc→sfu sfu→b→do +2→au		wrro 0 3 q510
exin03	h	exin04
alu→*→do au→a→ao,au doh→edb +2→au		wrro 0 q512
exin04 | csrsw4 | w		exin05
ag40→*→au alu→*d→do at→*p→au au→ao dol→edb		wrro 0 2 q514
exin05	h	exin06

sense int. pend. diag. (rs ↓ ad1)

au→a→ao,au ipb→sfu edb→di,irf +2→au rs32 0→sh→sl	trin
	0
	r002
spidr1 | exrx03 | w	spidr2
au→a→ao,au,pc irf→ird sfu→b1→r2 +2→au	irin
	0
	1
	r004
spidr2 | | h	bcnop2

store mode diag. (rs ↓ ad1)

sh→b→alu 0→alu msb→sfu	nacc
	1
	r102
smidr1 | mulsw3 |	mvisi2

sel. opr. sp. diag. (rs ↓ ad1)

ag0→a→sfu au→pc sfu→b→at xxx00xx→etmssrr	nacc
	0
	r105
sosdr0 | | |	start1

bus control diag. (rs ↓ ad1)

ag10→b→alu 0→alu	nacc
	4
	r202
bcdrs1 | bsdrs1 |	bcdrs2
au→pc dilz→*→au rb→*→au	nacc
	0
	r204
bcdrs2 | mvisi2 |	bcdrs3
alu→b→alu au→a→alu,sfu	nacc
	5
	r206
bcdrs3 | bsdrs3 |	bcdrs4
alu→a→at,au sfu→bsb 0→au	nacc
	0
	r208
bcdrs4 | bsdrs4 |	bcdrs5
at→*→ao,au doh→edb r2(1)→b→alu,do 0→alu 0→au	swpg
	1
	r210
bcdrs5 | |	bcdrs6
alu→*→do au→a→ao,au dol→edb +2→au	swpg
	0
	r212
bcdrs6 | pack18 |	bcdrs7
edb→irf exb→sfu pc→a→ao,au +2→au	trin
	0
	r214
bcdrs7 | | w	bsdrs8

bus sense diag. (rs ↓ ad1)

ag10→b→alu 0→alu	nacc
	4
	1
	r302
bsdrs1 | |	bsdrs2
au→pc dilz→*→au rb→*→au	nacc
	0
	r304
bsdrs2 | mvisi2 |	bsdrs3
alu→b→alu au→a→alu,sfu	nacc
	5
	1
	r306
bsdrs3 | |	bsdrs4
alu→a→at,au sfu→bsb 0→au	nacc
	0
	1
	r308
bsdrs4 | |	bsdrs5
at→a→au au→ao edb→di sfu→b1→sh1,dt 0→au	srpg
	0
	r310
bsdrs5 | spsw04 |	bsdrs6
au→ao edb→dil exb→sfu pc→a→au −2→au	srpg
	0
	r312
bsdrs6 | trss11 |	bsdrs7
edb→irf di→ad→r2(1) pc→ap→ao,au +2→au	trin
	0
	r314
bsdrs7 | | w	bsdrs8

au→a→ao,au,pc irf→ird sfu→b1→r2 +2→au	irin
	0
	r401
bsdrs8 | spidr2 | h	bsdrs9
edb→di,irf sfu→cc	frin
	0
	r403
bsdrs9 | |	slars4

load mim diag. (rs ↓ ad1)

ag0→ad→dt au→ap→ao,au dilz→b→at +2→au	irin
	0
	r406
mims01 | adssb1 | h	mims02
alu→dt au→pc dilz→b→au edb→di,irf rb→a→au	frin
	0
	r408
mims02 | bxhr×2 |	mims03
au→ao edb→dil	trop
	0
	r410
mims03 | cdsr06 | b	mims04
di→a→pur,sfu	nacc
	0
	r412
mims04 | unpk07 |	mims05
irf→ird pc→a→ao,au sfu→mim +2→au	irin
	0
	r414
mims05 | | h	bcnop2

machine check w internal (int ↓ a0)

ag30→a→ao,at,sl edb→di exb→sfu 0011101→etmssrr	srak
	0
	r502
mciin1 | mcin01 |	mciin2
ag08→ad→sfu at→ap→ao,au edb→dil sfu→b→dt 0→au	srak
	0
	r504
mciin2 | |	mciin3
di→b→sh sfu→bsb rs8 0→sh→sl	nacc
	0
	1
	r506
mciin3 | |	mciin6
dt→b1→sh1 ilc→sfu ls8 sh←0	nacc
	0
	r508
mciin6 | |	mciin4
at→a→au au→ao edb→di sfu→b1→sh1,dt 0→au	srmc
	0
	r510
mciin4 | spsw04 |	mciin5
at→a→ao,sl edb→dil pswh→sfu pswl→b→alu −4→alu 0011101→etmssrr	srmc
	1
	r512
mciin5 | ioin02 |	mciin7
alu→*→alu dt→*→alu	nacc
	1
	r514
mciin7 | |	mcin05

sel. reg. set diag. (rs ↓ ad1)

au→pc xxxxx01→etmssrr	nacc
	0
	r008
srsdr1 | | |	start1
ag0→a→au,sfu au→pc dilz→b→au xxxxx00→etmssrr	nacc
	0
	r011
srsdr2 | | |	start1
au→a→pc,sl xxxxx11→etmssrr	nacc
	0
	1
	r014
srsdr3 | | |	start1

sel. opr. sp. diag. (rs ↓ ad1)

au→pc ilc→sfu xxx01xx→etmssrr	nacc
	0
	1
	r108
sosdr1 | | |	start1
ag80000→a→at, sfu au→pc xxx10xx→etmssrr	nacc
	0
	1
	r111
sosdr2 | | |	start1
alu→a→alu au→pc sfu→b→alu xxx11xx→etmssrr	nacc
	1
	1
	r114
sosdr3 | | |	start1

rs — **ad1** — **stcm**

ag100→a→at,dt au→pc r1→b→sh sfu→bsb rs8 0→sh	nacc
	0
	1
	s002
stcm01	stcm02

au→pc dilz→*→au rb→*→au	nacc
	0
	s004
stcm02 | mvisi2	bc27

ire(12:13) = 00 **stcm03**
ire(12:13) = 01 **stcm10**
ire(12:13) = 10 **stcm10**
ire(12:13) = 11 **stcm08**

au→b→alu −1→alu	nacc
	1
	s007
stcm03	bc26

ire(14:15) = 00 **start1**
ire(14:15) = 01 **stcrx1**
ire(14:15) = 10 **stcm09**
ire(14:15) = 11 **sthrx1**

au→b→alu 0→alu	nacc
	1
	s010
stcm10	bc26

ire(14:15) = 00 **stcm04**
ire(14:15) = 01 **stcm12**
ire(14:15) = 10 **stcm12**
ire(14:15) = 11 **stcm13**

au→b→alu +1→alu	nacc
	1
	s013
stcm08	bc26

ire(14:15) = 00 **stcm04**
ire(14:15) = 01 **stcm12**
ire(14:15) = 10 **stcm12**
ire(14:15) = 11 **stcm13**

alu→b→alu +1→alu	nacc
	1
	1
	s101
stcm12	stcm04

alu→b→alu ipb→sfu +2→alu	nacc
	1
	1
	s104
stcm13	stcm04

alu→b→alu,ao edb→dil +2→alu	trow
	1
	s106
stcm04 | b	bc27

ire(12:13) = 01 **stcm05**
ire(12:13) = 10 **stcm06**
ire(12:13) = 11 **stcm07**

au→a→ao,au doh→edb sh→b→do +1→au	wrop
	0
	1
	s109
stcm06 | b	bc26

ire(14:15) = 00 **start1**
ire(14:15) = 01 **stcrx1**
ire(14:15) = 10 **stcm09**
ire(14:15) = 11 **sthrx2**

au→a→ao,au dol→edb sh→b→do +1→au	wrop
	0
	s112
stcm09 | b	start1

rs — **ad2** — **stm**

alu→dt au→ao dilz→*→au edb→dil rb→*→au	trow
	0
	s202
stmrs1 | lmrsw1 | b	stmrs3

au→a→ao,au doh→edb r1→b→do,dt +2→au	wrop
	0
	2
	s204
stmrs3 | w	bc17

r1 = r3 **sthrx2**
r1 ≠ r3 **stmrs2**

au→ap→ao,au dol→edb dt→ad→do pswl→b→alu −4→alu +2→au inc r1	wrop
	1
	1
	s207
stmrs2 | h	stmrs3

au→a→ao,au doh→edb sh→b→do +1→au	wrop
	0
	s211
stcm07 | stcm06 | b	stcm05

au→a→ao,au doh→edb r1→b→do +1→au	wrop
	0
	s213
stcm05 | b	bc26

ire(14:15) = 00 **start1**
ire(14:15) = 01 **stcrx1**
ire(14:15) = 10 **stcm09**
ire(14:15) = 11 **sthrx2**

rx — **ad2** — **st**

au→a→ao,at,au di→b→dt,sh edb→di pswh→sfu +2→au	trow
	0
	s302
strxw1 | adrxw3 | b	strxw2

at→a→au au→ao edb→di sfu→b1→sh1,dt 0→au	trow
	0
	s304
strxw2 | spsw04 | h	strxw3

au→a→ao,au doh→edb r1→b→do,dt +2→au	wrop
	0
	s306
strxw3 | stmrs3 | w	sthrx2

rx — **ad2** — **sth**

ag01→a→sfu au→ao edb→dil	trow
	0
	s309
sthrx1 | noxss5 | h	sthrx2

au→ao dol→edb r1→*→do	wrop
	0
	s311
sthrx2 | stcrx1 | h	start1

rx — **ad2** — **stc**

au→ao dol→edb r1→*→do	wrop
	0
	2
	s314
stcrx1	b | start1

s — **ad1** — **ssm**

au→pc dilz→a→at,au pswh→sfu sfu→b→alu,dt 0→alu 0→au	nacc
	1
	2
	s402
ssms01	ssms02

at→bp→au rb→a→au sfu→bd→alu,sh 0→alu rs16 0→sh→sl	nacc
	1
	2
	s404
ssms02	ssms03

alu→ad→sl au→b→ao,at edb→di pc→ap→au +2→au ls8 sh←sl←0	trop
	0
	s406
ssms03 | adrxh3 | b	ssms04

agb8→a→alu di→b→alu,sh rs8 0→sh→sl	nacc
	5
	1
	s408
ssms04	ssms05

at→b→au sl→a→at,au,sfu	nacc
	0
	s410
ssms05 | cvb16	ssms06

sfu→pswh	nacc
	0
	1
	s412
ssms06	bc20

alub = 0 **spmrr6**
alub ≠ 0 **spexc1**

rs — **ad1** — **store psw**

agffff8000→a→alu sfu→b→alu	nacc
	5
	s502
spsw01 | spmrr2	spsw02

au→bp→at dilz→*d→alu pc→ap→au rb→*d→alu +4→au	nacc
	1
	s504
spsw02	spsw03

alu→a→at,au au→pc pswh→sfu pswl→b→alu −4→alu +7→au	nacc
	1
	s506
spsw03	spsw04

at→a→au au→ao edb→di sfu→b1→sh1,dt 0→au	trow
	0
	3
	s508
spsw04	b | spsw05

au→a→ao,au di→ad→dt doh→edb sfu→b→do +2→au	wrop
	0
	s510
spsw05 | eudu09	w | spsw06

ag8c→ad→dt au→ap→ao,au dol→edb ilc→sfu sfu→b→do +2→au	wrop
	0
	s512
spsw06 | mvisi3	h | csrsw4

ss	ad2	trt
at→a→ao,sl au→b→dt edb→dil irf→ird		trop
		0
		t002
trtss1	b	trtss2
dibz→b→au dt→a→au		nacc
		0
		1
		t004
trtss2		trtss3
ag01→ad→sfu at→ap→au au→ao dt→b→at edb→di +1→au		trop
		0
		t006
trtss3	b	trtss4
alu→ad→dt at→ap→au au→bp→at di→bd→alu 0→alu 0→au		nacc
		1
		t008
trtss4		bc14

alu = 0 trtss6
alu ≠ 0 trtss5

alu→b→alu pc→a→au 0→alu 0→au		nacc
		3
		t011
trtss6		bc20

alub = 0 bcnop1
alub ≠ 0 trtss7

s	ad2	ts
agff→ad→do at→ap→ao di→b→alu dol→edb 0→alu		wrop
		2
		t014
tass01	b	start1

au→ap→ao,at,au edb→di +2→au		trow
		0
		1
		t101
trss13	b	trss10
at→ap→ao,au edb→di,irf sl→ad→dt +2→au		trin
		0
		t103
trss14	trss09 w	bcnop1

alu→b1→re2 irf→ird pc→a→ao,au +2→au		irin
		0
		t106
trtss7	h	trtss9
edb→di,irf sl→a7→re1		frin
		0
		t108
trtss9		ad1

alu→dt dt→b→alu −1→alu		nacc
		1
		t111
trtss5	pack06	bc20

alub = 0 trtss1
alub ≠ 0 trtss8

di→b1→re2 pc→a→ao,au sfu→cc +2→au		irin
		0
		t114
trtss8	h	trtss9

ss	ad2	tr
at→b→ao au→a→dt,sl edb→dil		trow
		0
		t202
trss01	b	bc14

alu = 0 trss10
alu ≠ 0 trss03

alu→a→ao dt→b→alu edb→irf −1→alu		trop
		1
		t205
trss05	b	trss04
alu→b→alu,sh at→ap→au −1→alu +1→au		nacc
		1
		t207
trss03		trss04
alu→dt au→ap→ao,au dibz→b→alu edb→dil sl→ad→alu +1→au		trow
		1
		t209
trss04	b	bc14

alu = 0 trss06
alu ≠ 0 trss05

si	ad2	tm
au→ap→ao,au,pc di→ad→alu irf→ird sfu→b→alu +2→au		irin
		2
		1
		t212
tmrxb1	h	tmrxb2
alu→a→alu edb→di,irf sfu→b→alu		frin
		3
		t214
tmrxb2		ad1

alu→b→ao,au dt→a→au		irop
		0
		t301
trss06	aissb5 b	trss07
dibz→b→alu edb→dil sl→a→alu		frop
		1
		t303
trss07		trss08
alu→a→ao,au edb→di sh→b→alu,au,dt 0→alu		trop
		1
		t305
trss08	apssb8 b	trss09
at→ap→ao,au edb→di,irf sl→ad→dt +2→au		trow
		0
		1
		t307
trss09	b	trss10
dibz→b→au dt→a→au		nacc
		0
		t309
trss10	trtss2	trss11
au→ao edb→dil exb→sfu pc→a→au −2→au		trop
		0
		3
		t311
trss11	b	trss12
alu→bd→alu at→ap→ao,au au→bp→at di→ad→do dol→edb −1→alu +1→au		wrop
		1
		t313
trss12	mvcss2 b	bc14

alu = 0 trss14
alu ≠ 0 trss13

rr	ad1	svc
pce→a→dt sfu→b→sh		nacc
		0
		t402
svcrr1		svcrr4
ag20→a→at,sl dt→b→alu pswh→sfu 0→alu 0011101→etmssrr		nacc
		1
		t404
svcrr4		svcrr2
ag88→ad→dt at→ap→ao,au doh→edb sfu→b→do +2→au		wrro
		0
		1
		t406
svcrr2	w	svcrr3
au→a→ao,au dol→edb ilc→sfu sfu→b→do +2→au rs32 0→sh→sl		wrro
		0
		t408
svcrr3	h	prin04

au→bp→at dol→edb dt→a→ao,au sfu→bd→do +2→au		wrro
		0
		t412
prin06	w	prin07
at→ap→au au→ao dol→edb sl→ad→do 0→au		wrro
		0
		t414
prin07	h	exin07

int	a0	null & comp. program
ag20→a→ao,at,au edb→di pswh→sfu +7→au		srak
		0
		1
		t502
prin01		prin08
ag8c→bd→alu at→bp→ao au→a→au,sl edb→dil 0→alu +1→au 0011101→etmssrr		srak
		1
		1
		t504
prin08		prin02
au→ap→ao,at,au di→ad→sl doh→edb sfu→b→do +2→au		wrro
		0
		t506
prin02	exin02 w	prin03
au→ap→ao,au di→ad→sl dol→edb ilc→sfu sfu→b→do +2→au		wrro
		0
		t508
prin03	exin03 h	prin09
alu→dt pswl→b→alu −4→alu ls8 sh←0		nacc
		1
		t510
prin09	spexc2	prin04
alu→*→do au→a→ao,au doh→edb +2→au		wrro
		0
		t512
prin04	csrsw4 w	prin05
ag40→*→au alu→*d→do at→*p→au au→ao dol→edb		wrro
		0
		t514
prin05	exin05 h	prin06

ss — **ad2** ↓ — **unpk**

alu→b→alu au→a→ao,au edb→di −1→alu −1→au			trop
			1
			u002
unpk01	pack01	b	unpk02
alu→dt di→a→sfut sfu→b→alu −1→alu			nacc
			1
			u004
unpk02	pack02		bc02

alu < 0 **unpk03**
alu ≥ 0 **unpk05**

ag0→ad→pur at→ap→ao,au au→bp→at dol→edb sfu→bd→do −1→au			wrop
			0
			u007
unpk05	pack10	b	bc06

alu < 0 **start1**
alu = 0 **unpk06**
alu > 0 **unpk09**

at→*→ao di→*d→dt edb→dil			trop
			0
			u010
unpk06	mvnss2	b	unpk07
di→a→pur,sfu			nacc
			0
			2
			u012
unpk07			unpk08
au→ao dol→edb pc→ap→au pur→ad→do −2→au			wrop
			0
			u014
unpk08		b	start1

ag0→ad→pur at→ap→ao,au au→bp→at dol→edb sfu→bd→do −1→au			wrop
			0
			u101
unpk03	pack10	b	bc02

alu < 0 **start1**
alu ≥ 0 **unpk04**

alu→b→alu au→ap→ao,au dol→edb pur→ad→do −1→alu −1→au			wrop
			1
			u104
unpk04		b	bc05

alu > 0 **unpk04**
alu ≤ 0 **start1**

alu→dt at→a→ao,au au→bp→at dt→bd→alu edb→di −1→alu −1→au			trop
			1
			2
			u107
unpk09		b	unpk10
di→a→pur,sfu			nacc
			0
			u109
unpk10	unpk07		unpk11
alu→dt at→ap→ao,au au→bp→at dol→edb dt→bd→alu pur→ad→do −2→alu; −1→au			wrop
			1
			u111
unpk11		b	bc02

alu < 0 **unpk13**
alu ≥ 0 **unpk12**

ssbu — **ad1** ↓ — **addr**

au→a→ao,au l1→sfu sfu→b→alu,sh 0→alu +2→au			irin
			1
			u202
aussb1	amssb1	h	aussb2
alu→dt au→pc dilz→a→au edb→di sfu→b→alu,at +1→alu 0→au			frin
			1
			u204
aussb2			aussb3
alu→bd→sh au→bp→au rb→a→au rs1 0→sh→sl			nacc
			0
			u206
aussb3			aussb4
dt→a→alu sh→b→alu			nacc
			4
			1
			u208
aussb4			aussb5
at→*→au au→*→ao,au dt→bd→sh			irop
			0
			u210
aussb5		b	apssb5

ag0→ad→pur au→ap→ao,au dol→edb pur→b→do −1→au			wrop
			0
			u213
unpk12	unpk13	b	bc06

alu < 0 **start1**
alu = 0 **unpk06**
alu > 0 **unpk09**

rx — **ad1** ↓ — **fpxs**

dilz→*→au x2→*→au			nacc
			0
			u302
fpxs01	fprx01		fpxs02
ag01→bd→alu au→bp→au rb→a→au 0→alu			nacc
			1
			u304
fpxs02	fprx02		fpxs03
agf0→a→ao,sl alu→dt edb→di			trro
			0
			u306
fpxs03	fprr01	h	fpxs04
ag06→a→sfu dile→b→alu,sh 0→alu ls4 sh←sl←0			nacc
			1
			u308
fpxs04	fprr02		fpxs05
au→*→at,au dt→*→au sfu→bsb			nacc
			0
			u310
fpxs05	fprx05		bc14

alu = 0 **opexc1**
alu ≠ 0 **fpxs06**

ag0→ad→pur au→ap→ao,au dol→edb pur→b→do −1→au			wrop
			0
			1
			u313
unpk13		b	bc02

alu < 0 **start1**
alu ≥ 0 **unpk04**

ag10→a→au au→ao edb→di ire→sfu sh→b→au			trow
			0s
			u401
fpxs06	fprx06	b	fpxs07
at→a→ao,dt dt→bd→alu edb→di 0→alu			trow
			1
			u403
fpxs07		b	fpxs08
au→bp→ao,at dol→edb pc→a→au sfu→bd→do +2→au			swmc
			0
			u405
fpxs08			fpxs09
at→bp→ao au→pc dol→edb dt→a→at,au sfu→bd→do 0→au			swmc
			0
			u407
fpxs09			fpxs10
au→a→at edb→di sh→b→ao			srmc
			0
			u409
fpxs10			fpxs11
ag10→a→au edb→di sh→b→ao,au			srmc
			0
			u411
fpxs11			bc14

alu = 0 **fpxs12**
alu ≠ 0 **fpxs13**

at→*→ao di→*→do dol→edb			wrop
			0
			u414
fpxs12		h	fprr07

rx — **ad1** ↓ — **fpxl**

dilz→*→au x2→*→au			nacc
			0
			u502
fpxl01	fprx01		fpxl02
ag03→bd→alu au→bp→au rb→a→au 0→alu			nacc
			1
			u504
fpxl02	fprxl2		fpxs03

rr — **ad2** ↓ — **fprrop**

au→ao edb→dil			srmc
			0
			u507
fprr07	cdsr06		fprr08
au→ao edb→dil			srmc
			0
			u509
fprr08	cdsr06		fprr09
ag0c→b→alu di→a→alu,pur,sfu			nacc
			5
			u511
fprr09	fpcr09		fprr10

alu→b→alu at→ap→ao,au di→ad→do dol→edb −1→alu +2→au			wrop
			1
			u514
fpxs13		h	fpxs10

none ↓ ad1	operation exception
agba→a→au msb→sfu +6→au	nacc
	0
	1
	w002
opexc1	opex02
ag10→ad→alu pswh→sfu sfu→b→alu,sh ls8 sh←0	nacc
	5
	1
	w004
opex02	opex03
ag01→a→sl pswl→b→alu −4→alu 0011101→etmssrr	nacc
	1
	w006
opex03	bc20
alub = 0	**opex04**
alub ≠ 0	**opexc2**

au→a→ao,au doh→edb sfu→b→do,dt +2→au ls16 sh←0	wrro
	0
	w009
opex04 w	opex05
agffff8000→b→ex au→ap→ao,au dol→edb dt→ad→do ire→sfu +6→au ls1 sh←0	wrro
	0
	w011
opex05 h	opex06
au→ap→ao,au dol→edb ilc→sfu sfu→b→do,dt,sh −4→au rs8 0→sh	wrro
	0
	w013
opex06 h	bc22
sh(0) = 0	**opex07**
sh(0) = 1	**opex30**

ag440000→bd→sh rs16 0→sh→sl	nacc
	0
	1
	w101
opex30	opex07
alu→ad→alu sfu→bd→alu rs32 0→sh→sl	nacc
	1
	w103
opex07	opex08
ag3f00→b→at,sh alu→ad→do au→ap→ao,au doh→edb +2→au ls2 sl←sh←0	wrro
	0
	w105
opex08 w	opex09
alu→bd→do at→bp→au au→ao dol→edb sl→a→au	wrro
	0
	w107
opex09 h	opex10
ag80000→ad→sfu au→ap→ao,au dt→b→at,sh edb→di +2→au rs8 0→sh	trro
	0
	w109
opex10 w	opex11
ag40→*→alu at→*→au au→ao edb→dil ex→*→au sh→*→alu	trro
	5
	w111
opex11 h	opex12
di→a→ao,au dt→b→sh +2→au rs16 0→sh→sl sfu→pswh	irin
	0
	w113
opex12 h	bc15
(alu = 0)(au < 0)	**opex32**
(alu = 0)(au ≥ 0)	**opex24**
(alu ≠ 0)(au < 0)	**opex13**
(alu ≠ 0)(au ≥ 0)	**opex18**

au→*→at edb→irf xxx01xx→etmssrr	frin
	0
	1
	w201
opex13	opex14
edb→di pc→a→ao,au +2→au	trop
	0
	1
	w203
opex14 h	opex15
at→ap→au dilz→b→alu x2→ad→alu +2→au	nacc
	1
	w205
opex15 cead06	opex16
alu→*→alu at→a→ao,au,pc irf→ird rb→*→alu +2→au xxx11xx→etmssrr	irin
	1
	w207
opex16 h	opex17
alu→a→re1 edb→di,irf	frin
	0
	w209
opex17	ad1

au→a→ao,au,pc di→b→alu irf→ird 0→alu +2→au	irin
	1
	w212
opex29 bcnop1 h	opex31
edb→di,irf xxx11xx→etmssrr	frin
	0
	w214
opex31	ad1

agfff00000→b→alu au→*→at edb→irf sl→a→alu	frin
	5
	w301
opex24	opex25
agb2200000→* →alu alu→*→alu xxx01xx→etmssrr	nacc
	4
	w303
opex25	opex26
agff000000→b→alu sl→a→alu	nacc
	5
	w305
opex26	bc14
alu = 0	**opex29**
alu ≠ 0	**opex27**

agb1000000→*→alu alu→*→alu pc→*p→ao	irop
	4
	w308
opex27 h	opex33
edb→di	frop
	0
	w310
opex33	bc14
alu = 0	**opex15**
alu ≠ 0	**opex28**

dilz→b→alu 0→alu	nacc
	1
	w313
opex28	opex16

au→*→at edb→irf xxx01xx→etmssrr	frin
	0
	w401
opex18 opex13	opex19
edb→di pc→a→ao,au +2→au	trop
	0
	w403
opex19 opex14 w	opex20
au→ao dilz→b→alu edb→di 0→alu	trop
	1
	w405
opex20 h	opex21
alu→*→alu rb→*→alu xxx11xx→etmssrr	nacc
	1
	w407
opex21	opex22
alu→dt at→ap→ao,au,pc dilz→*→alu irf→ird rb→*→alu +2→au	irin
	1
	w409
opex22 h	opex23
alu→b→re2 dt→a→re1 edb→di,irf	frin
	0
	w411
opex23	ad1

edb→di,irf	frin
	0
	w414
opex32 bcnop2	bcnop1

int ↓ a0	trace
ag80→a→sl pswh→sfu 0011101→etmssrr	nacc
	0
	w502
trace2	trace3
pc→b→alu −2→alu	nacc
	1
	w504
trace3	trace4
ag28→a→ao,at,au doh→edb sfu→b→do +2→au	wrro
	0
	w506
trace4 opexc2 w	trace5
ag8c→ad→dt au→ap→ao,au dol→edb ilc→sfu sfu→b→do +2→au	wrro
	0
	w508
trace5 mvisi3 h	prin04

ag10→ad→alu pswh→sfu sfu→b→alu,sh ls8 sh←0	nacc
	5
	w511
popex2 opex02	popex3
ag02→a→sl pswl→b→alu −4→alu 0011101→etmssrr	nacc
	1
	w513
popex3	bc20
alub = 0	**opex04**
alub ≠ 0	**opexc2**

rs	a1 ↓		eu dump diag.
ag100→a→ao,au au→pc doh→edb dt→b→do +2→au			wrro
			0
			x002
eudu01		w	eudu02
au→a→ao,au dol→edb dt→*→do +2→au			wrro
			0
			x004
eudu02	cvd20	h	eudu03
au→a→ao,au doh→edb pc→b→do,dt +2→au			wrro
			0
			1
			x006
eudu03		w	eudu04
au→a→ao,au dol→edb dt→*→do +2→au			wrro
			0
			x008
eudu04	cvd20	h	eudu05
au→a→ao,au di→*→do,dt doh→edb +2→au			wrro
			0
			1
			x010
eudu05		w	eudu06
au→a→ao,au dol→edb dt→*→do +2→au			wrro
			0
			x012
eudu06	cvd20	h	eudu07
alu→*→do au→a→ao,au doh→edb +2→au			wrro
			0
			x014
eudu07	csrsw4	w	eudu08

alu→*→do au→a→ao,au dol→edb +2→au			wrro
			0
			x102
eudu08	pack18	h	eudu09
au→a→ao,au di→ad→dt doh→edb sfu→b→do +2→au			wrro
			0
			3
			x104
eudu09		w	eudu10
ag8c→ad→dt au→ap→ao,au dol→edb ilc→sfu sfu→b→do +2→au			wrro
			0
			x106
eudu10	mvisi3	h	eudu11
au→a→ao,au doh→edb sh→b→do,dt +2→au			wrro
			0
			1
			x108
eudu11		w	eudu12
au→a→ao,au dol→edb dt→*→do +2→au			wrro
			0
			x110
eudu12	cvd20	h	eudu13
au→ap→ao,au doh→edb mim→sfu sl→ad→do,dt +2→au			wrro
			0
			1
			x112
eudu13		w	eudu14
au→*p→ao,au dol→edb dt→*→do sfu→b1→sh1 ls8 sh←0 +2→au			wrro
			0
			x114
eudu14		h	eudu36

agf0→a→alu au→b→alu bsb→sfu			nacc
			5
			x202
eudu36	spkas3		eudu37
sfu→b1→sh1 ls8 sh←0			nacc
			0
			1
			x204
eudu37			eudu46
sfu→b1→sh1 ls8 sh←0			nacc
			0
			x206
eudu38	eudu37		eudu39
alu→b→alu ipb→sfu +2→alu			nacc
			1
			x208
eudu39	stcm13		eudu40
sfu→b1→sh1			nacc
			0
			x210
eudu40			eudu41
au→a→ao,au doh→edb sh→b→do,dt +2→au			wrro
			0
			x212
eudu41	eudu11	w	eudu42
au→a→ao,au dol→edb dt→*→do +2→au			wrro
			0
			x214
eudu42	cvd20	h	eudu15

au→a→ao,au di→ad→dt doh→edb sfu→b→do +2→au			wrro
			0
			x302
eudu15	eudu09	w	eudu16
au→ap→ao,au dol→edb ilc→sfu pur→b→do +2→au			wrro
			0
			x304
eudu16		h	eudu17
ag8c→ad→dt au→ap→ao,au dol→edb ilc→sfu sfu→b→do +2→au			wrro
			0
			x306
eudu17	mvisi3	w	eudu18
ird→ire ire→sfu pswl→b→dt			nacc
			0
			3
			x308
eudu18			eudu34
alu→b→alu,sh irf→ird 0→alu rs16 0→sh→sl			nacc
			5
			x310
eudu34	spkas6		eudu19
ag8c→ad→dt au→ap→ao,au dol→edb ilc→sfu sfu→b→do +2→au			wrro
			0
			x312
eudu19	mvisi3	h	eudu20
ird→ire ire→sfu pswl→b→dt			nacc
			0
			x314
eudu20	eudu18		eudu21

ag8c→ad→dt au→ap→ao,au dol→edb ilc→sfu sfu→b→do +2→au			wrro
			0
			x402
eudu21	mvisi3	w	eudu22
ird→ire ire→sfu pswl→b→dt			nacc
			0
			x404
eudu22	eudu18		eudu23
ag8c→ad→dt au→ap→ao,au dol→edb ilc→sfu sfu→b→do +2→au			wrro
			0
			x406
eudu23	mvisi3	h	eudu24
at→b→do,dt au→ap→ao,au doh→edb pswh→sfu +2→au			wrro
			0
			x408
eudu24		w	eudu25
au→a→ao,au dol→edb dt→*→do +2→au			wrro
			0
			x410
eudu25	cvd20	h	eudu26
au→a→ao,au di→ad→dt doh→edb sfu→b→do +2→au			wrro
			0
			x412
eudu26	eudu09	w	eudu27
ag8c→ad→dt au→ap→ao,au dol→edb ilc→sfu sfu→b→do +2→au			wrro
			0
			x414
eudu27	mvisi3	h	eudu35

ird→ire ire→sfu pswl→b→dt			nacc
			0
			x502
eudu35	eudu18		eudu28
ag0f→ad→sfu au→ap→ao,au doh→edb dt→b→do +2→au			wrro
			0
			x504
eudu28		w	eudu29
au→a→ao,au dol→edb dt→*→do +2→au			wrro
			0
			x506
eudu29	cvd20	h	eudu30
au→a→ao,au doh→edb ex→b→do,dt sfu→ire +2→au			wrro
			0
			x508
eudu30		w	eudu31
au→a→ao,au dol→edb dt→*→do +2→au			wrro
			0
			x510
eudu31	cvd20	h	eudu32
au→a→ao,au doh→edb pc→b→do,dt +2→au			wrro
			0
			x512
eudu32	eudu03	w	eudu33
au→a→ao,au dol→edb dt→*→do +2→au			wrro
			0
			x514
eudu33	cvd20	h	eudu43

rs	ad1 ↓	calc. ea diag.
ag0→a→sfu r2→b→alu,ao +2→alu		irop
		1
		y002
cead01	h	cead02
alu→dt au→pc dilz→b→au edb→di,irf rb→a→au		frop
		0
		y004
cead02	bxhrx2	cead03
au→bp→at di→a→alu,au di→bd→alu irf→ird 0→au		nacc
		1
		y006
cead03		cead04
au→b→alu dt→a→au 0→alu 0→au		nacc
		1
		y008
cead04	ddr17	bc09
(alu<0)(au<0)		**cead11**
(alu≥0)(au≥0)		**cead10**
(alu≥0)(au<0)		**cead17**
(alu<0)(au≥0)		**cead05**
edb→irf pc→a→ao,au sfu→cc +2→au		trin
		0
		1
		y011
cead10	w	bcnop1

au→a→ao,au di→b→dt,sh edb→di +2→au	trop
	0
	y101
cead11 cdsr03 w	cead12
ag02→a→sfu au→ao dilz→b→alu edb→di 0→alu	trop
	1
	y103
cead12 h	cead13
alu→*→alu at→ap→au rb→*→alu +7→au	nacc
	1
	y105
cead13	cead14
alu→dt au→ao dilz→*→au edb→dil rb→*→au	trow
	0
	y107
cead14 lmrsw1 b	cead15
at→ap→ao,au au→b→alu doh→edb dt→ad→do 0→alu +2→au	wrop
	1
	y109
cead15 w	cead16
au→a→ao,au dol→edb dt→*→do +2→au	wrop
	0
	y111
cead16 cvd20 h	cead08

agb1000000→a→alu alu→b→alu au→ap→ao,au ird→ire +2→au	irop
	4
	y201
cead05 cead20 h	cead23
ag01→a→sfu edb→di,irf sfu→b→sh rs16 0→sh→sl	frop
	0
	y203
cead23 ipks02	cead06
at→ap→au dilz→b→alu x2→ad→alu +2→au	nacc
	1
	1
	y205
cead06	cead07
alu→*→alu at→*p→au au→ao edb→irf rb→*→alu 0→au	trow
	1
	y207
cead07 h	cead08
alu→*→do au→a→ao,au doh→edb sfu→cc +2→au	wrop
	0
	y209
cead08 w	cead09
alu→*→do au→a→ao,au dol→edb +2→au	wrop
	0
	y211
cead09 pack18 h	start1

agfff00000→a→alu di→b→alu	nacc
	5
	y301
cead17	cead18
agb2200000→a→alu alu→b→alu	nacc
	4
	y303
cead18	cead19
agff000000→a→alu di→b→alu	nacc
	5
	y305
cead19	bc14
alu = 0	**cead10**
alu ≠ 0	**cead20**

agb1000000→a→alu alu→b→alu au→ap→ao,au ird→ire +2→au	irop
	4
	1
	y308
cead20 h	cead21
ag01→a→sfu edb→di,irf sfu→b→sh rs16 0→sh→sl	frop
	0
	y310
cead21 ipks02	bc14
alu = 0	**cead06**
alu ≠ 0	**cead22**

di→b1→sh1	nacc
	0
	y313
mkin06 icmr05	bc14
alu = 0	**mkin07**
alu ≠ 0	**mkin08**

int	a0 ↓	mc68000
ag40→a→au,sl lvl→sfu 0→au 0011101→etmssrr		nacc
		0
		y402
mkin01		mkin02
ag08→a→alu,au sfu→b→alu,au,sh ls1 sl←sh←0		nacc
		1s
		y404
mkin02		mkin10
au→a→au au→b→au		nacc
		0
		y406
mkin10		mkin03
ag40→a→au au→b→ao,at edb→di 0→au		srak
		0
		y408
mkin03		mkin04
ag10→a→alu alu→b→alu,sh at→*p→ao edb→dil exb→sfu		srak
		1
		y410
mkin04		mkin05
alu→ad→sl au→ap→at,au pswh→sfu sfu→b→alu +3→alu −8→au ls8 sh←sl←0		nacc
		5
		y412
mkin05		mkin06
au→a→ao,au doh→edb sfu→b→do,dt +2→au		wrro
		0
		y414
mkin09	w	ioin05

int	a0 ↓	i/o
ag40→a→ao,at,au edb→di −8→au		srak
		0
		y502
ioin01		ioin02
at→a→ao,sl edb→dil pswh→sfu pswl→b→alu −4→alu 0011101→etmssrr		srak
		1
		1
		y504
ioin02		ioin04
au→ap→ao,au di→ad→sl doh→edb sfu→b→do,dt +2→au		wrro
		0
		y506
ioin04	w	ioin05
au→a→ao,au dol→edb dt→*→do +2→au		wrro
		0
		y508
ioin05	cvd20 h	ioin06
alu→*→do au→a→ao,au doh→edb +2→au		wrro
		0
		y510
ioin06	csrsw4 w	ioin07
agb8→b→alu,au,dt alu→ad→do at→ap→au au→ao dol→edb +2→alu		wrro
		1s
		y512
ioin07	h	exin13

at→a→au dilz→b→alu 0→alu +2→au	nacc
	1
	y014
cead22	cead07

alu→dt pswl→b→alu −4→alu ls8 sh←0	nacc
	1
	y114
mkin07 spexc2	mkin09

pswl→b→alu −4→alu rs32 0→sh→sl	nacc
	1
	y214
mkin08	mkin09

ss — ad1 — decimal

agf2→a→ao,sl dilz→*→at edb→di			trro
			0
			z002
decm01		h	decm02
at→bp→au au→ao dile→bd→alu,sh rb→a→au 0→alu ls4 sh←sl←0			irin
			1
			z004
decm02		h	decm03
ag06→ad→sfu au→b→dt edb→di pc→ap→au +4→au			frin
			0
			z006
decm03			bc14

alu = 0 **opexc1**
alu ≠ 0 **decm04**

ag10→a→at,au au→pc sfu→bsb sh→b→au			nacc
			0
			z009
decm04			decm05
au→ao doh→edb dt→*→do ire→sfu rs32 0→sh→sl			swmc
			0
			z011
decm05			decm06
at→*p→au au→*p→ao,au dol→edb dt→ad→do sfu→bd→sh			swmc
			0
			z013
decm06			decm07

dilz→*→alu msb→sfu rb→*→alu ls16 sh←0			nacc
			1
			z102
decm07			decm08
alu→a→do au→ao doh→edb pswh→sfu sfu→b→dt			swmc
			0
			z104
decm08			decm09
alu→a→do au→ao dol→edb sfu→be→dt			swmc
			0
			z106
decm09			decm10
doh→edb sh→b→do sl→a→ao,at,au 0→au			swmc
			0
			z108
decm10			decm11
ag10→*→au at→*p→ao,au dol→edb dt→*d→do			swmc
			0
			z110
decm11			decm12
at→a→ao,au au→b→at edb→di 0→au			srmc
			0
			z112
decm12			decm13
au→ao edb→dil			srmc
			0
			z114
decm13	cdsr06		decm14

dile→b→alu,sh 0→alu rs8 0→sh			nacc
			1
			z201
decm14			decm15
di→ad→sfu pc→ap→au sh→b→alu,dt +2→alu +2→au			nacc
			5
			z203
decm15			bc02

alu < 0 **decm18**
alu ≥ 0 **decm16**

at→a→ao,au au→pc edb→di irf→ird 0→au rs32 0→sh→sl			srmc
			0
			1
			z206
decm18			decm19
au→ao edb→dil exb→sfu pc→a→au −2→au			srmc
			0
			z208
decm19	trss11		bc16

sl(31) = 0 **decm20**
sl(31) = 1 **decm21**

di→a→ao,au edb→di +2→au			trop
			0
			1
			z211
decm20		h	opexc1

di→a→ao,au edb→di +2→au			trow
			0
			z214
decm21	decm20	h	opexc1

dt→ad→sl pc→ap→ao,au sfu→cc +2→au			irin
			0
			z301
decm16		h	bc14

alu = 0 **decm22**
alu ≠ 0 **decm17**

ag01→a→sfu edb→di,irf sfu→b→sh rs16 0→sh→sl			frin
			0
			z304
decm22	ipks02		bc16

sl(31) = 0 **bcnop1**
sl(31) = 1 **decm23**

at→a→ao,au au→pc edb→di irf→ird 0→au rs32 0→sh→sl			srmc
			0
			z307
decm23	decm18		decm24
au→ao edb→dil exb→sfu pc→a→au −2→au			srmc
			0
			z309
decm24	trss11		decm25
di→ad→re1 edb→di,irf pc→ap→ao,au +2→au			trin
			0
			z311
decm25		h	ad1

dt→ad→sl edb→irf pswh→sfu pswl→b→alu −4→alu 0011101→etmssrr			frin
			1
			z314
decm17			opexc2

rs — ad1 — bxle

au→ap→ao,au r1→ad→alu r2→b→alu +2→au			irin
			1
			z402
bxlex1	bxhrx1	h	bxlex2
alu→dt au→pc dilz→b→au edb→di,irf rb→a→au			frin
			0
			z404
bxlex2	bxhrx2		bxlex3
alu→b→alu,sh au→ap→ao,au irf→ird r2(1)→ad→alu +2→au			irin
			3
			z406
bxlex3	bxhrx3	h	bxlex4
edb→di,irf sh→b→r1			frin
			0
			z408
bxlex4	slrrs2		bc03

aluv = 0 **bxlex5**
aluv = 1 **bxhrx6**

agb8→a→alu di→b→alu,sh rs8 0→sh→sl			nacc
			5
			z411
bxlex5	ssms04		bc05

alu > 0 **bcrrw2**
alu ≤ 0 **bctrw3**

rs — ad1 — mc68000 reset diag.

au→pc dilz→*→au rb→*→au			rest
			0
			z414
mkrst1	mvisi2		start1

ipl — a01 — ipl

ag80000→a→ao,at ag80000→a→sl edb→di 0011101→etmssrr			srak
			0
			z502
ipl01			ipl02
ag08→ad→sfu at→ap→ao,au edb→dil pswl→b→alu −4→alu 0→au			srak
			1
			z504
ipl02	mcin02		ipl03
r1→b→sh sfu→bsb			nacc
			0
			z506
ipl03	iskrr2		ipl04
agb80840ff→b →alu au→ap→ao,au di→ad→alu,dt,sfu edb→di +2→au			srmc
			5
			z508
ipl04	lpsw08		ipl05
ag80000→a→alu alu→b→alu au→ao edb→dil			srmc
			6
			z510
ipl05	exin10		lpsw10

A P P E N D I X

E

IBM Micro/370 Microprocessor General Information

24 April 1986
Joseph P. Higham
Thomas J. Watson Research Center

Highlights

- Single-chip nMOS 32-bit microprocessor
- Executes 102 System/370 instructions on-chip
- System/370 interruption handling
- 10-MHz worst case clock rate (corresponds to a 200ns processor cycle time)
- More than 250,000 System/370 instructions per second at 10-MHz
- Virtual storage support
- Coprocessor support
- Overlapped instruction fetch, decode, and execution
- 32-bit internal data path
- Motorola MC68000/MC68008 bus compatibility
- Nonmultiplexed 32-bit address and data buses
- Direct program control of pins

- Dynamically selected 1-, 2-, or 4-byte data transfers
- Asynchronous bus interface
- External bus arbitration
- Single +5V supply
- Power dissipation: 3 watts normal, 4 watts worst case
- 10mm × 10mm chip with C4s on double row of pads, packaged on 36mm single-chip module with 171 pins
- 106 signal and I/O pins
- 2 micron, silicon-gate nMOS process (single poly, double metal)
- 200,000 transistor sites (93,000 actual)
- 94,208 (92K) bit on-chip control store

MC68000, MC68008, MC6800, M68000, and M6800 are trademarks of Motorola Semiconductor Products Inc.

Overview

Micro/370 is a 32-bit, single-chip, System/370 microprocessor. It executes 102 System/370 instructions. (One of the 102 instructions, Diagnose, has 14 independent suboperations.) The chip includes coprocessor interfaces for the execution of an additional 60 System/370 instructions. Micro/370 provides a special operating mode called "Dual mode" to allow emulation of unimplemented instructions outside the user memory space. Micro/370 can be used without coprocessors. The external interface allows concurrent operation of 8-, 16-, and 32-bit devices, with an address space of up to 4 gigabytes. The pin interface is logically and electrically compatible with Motorola MC68000 and MC6800 peripheral devices. Address and data buses are extended to 32 bits. Micro/370 features a special set of pin-programmable, pin-readable bus cycles that are directly accessible to the System/370 program via the System/370 Diagnose instructions. These are meant to be used for control applications. This chip is implemented in a 2-micron (1.8-micron effective channel length), silicon-gate process to achieve a high level of integration.

Micro/370 Component Description

The Micro/370 microprocessor is a single-chip, 32-bit processor capable of addressing 4 gigabytes of real memory. Micro/370 directly executes 102 System/370 instructions. The processor

contains all sequencing and processing controls for instruction execution, exception processing, interruption action, coprocessor communication, external storage access, instruction emulation, and other control-related functions.

Figure E.9 (page 320) compares the Micro/370 instruction subset with the full set of System/370 instructions. Of the 104 standard instructions, Micro/370 implements 91 of the general instructions and control instructions. Figure E.10 (page 321) lists the instructions executed by Micro/370. Micro/370 supports coprocessor execution of the 9 decimal instructions and the 51 floating-point and extended-precision floating-point instructions. With a 10-MHz clock, Micro/370 executes more than 250,000 instructions per second.

Sample instruction execution times:

Instruction	Nanoseconds
Add (register-to-register)	400
Add (memory-to-register)	1,300
Branch on Condition (branch not taken)	800
Branch on Condition (branch taken)	1,100
Load (register-to-register)	400
Load (memory-to-register)	1,300
Store (register-to-memory)	2,400
Left Shift (logical)	800
Supervisor Call	5,100
Test Under Mask	1,100

Assumptions:
1. 10-MHz part
2. 400 nanosecond MC68000 memory-access write cycle
3. 300 nanosecond MC68000 memory-access read cycle
4. 32-bit memory system

Micro/370 supports bus protocols compatible with the Motorola MC68000 component bus. This means Micro/370 can be used directly with MC68000 and MC6800 peripheral parts.

Micro/370 external signals can be divided into ten groups: address, data, asynchronous bus control, bus arbitration control, interrupt control, processor status, indicators, timing, MC6800 peripheral control, and miscellaneous control. An eleventh group, testing, provides control for chip testing and verification. Descriptions of the Micro/370 bus signals are in a following section.

Figure E.1 and figure E.2 (page 307) show the Micro/370 external signals.

Micro/370 is implemented in 2-micron nMOS with single-level poly and two levels of metal. The chip image is 10mm by 10mm with a double row of C4 pads around the edge. Of the 171 module pins, 106 are Micro/370 I/O signals. Figure E.11 (page 322) diagrams the pinout for the module. Only a single, +5V supply voltage is necessary. A list of Micro/370 pin assignments appears in figure E.12 (page 323).

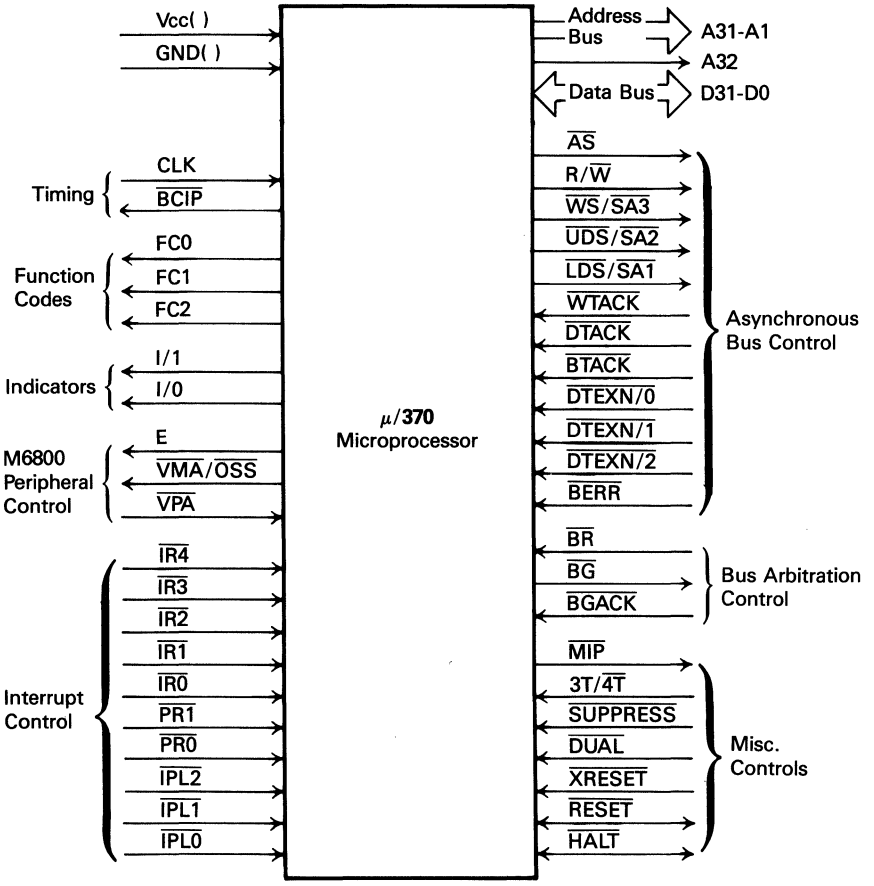


Figure E.1 Micro/370 signals

Package Information

Micro/370 is mounted with C4s on a 36mm ceramic substrate, the Single Chip Module. This substrate has 171 pins, substrate P/N 6244060. An IBM Burlington thermal grease cap is used to aid heat dissipation, thermal cap P/N 6837105.

Electrical Characteristics of the Micro/370 Module:

- Power Dissipation = 3 watts nominal, 4 watts worst case
- Vdd = 5.0 volts ± 10%

Micro/370 Signal Descriptions (Figure E.2)

Address Bus

A31 is the most significant bit of a 32-bit byte address. A31–A1 address halfword data (upper and lower data strobe determine upper or lower byte addresses). This allows a 4-gigabyte address space.

During a Motorola interrupt acknowledge cycle (function code 7), bits A3–A1 contain the interrupt level.

During Micro/370 service cycles (function code 7), bits A3–A1 are zero and bits A31–A4 carry 28 bits of addressing or command information. The service address bits specify the type of service cycle.

Real Address Tag

Real Address Tag is set (A32=1) for low-memory address accesses (System/370 assigned storage locations) and service cycles.

Data Bus

D31–D0 provide a 32-bit data path for Micro/370 bus operations. Data transfers can have data widths of 1 byte, 2 bytes (halfword), or 4 bytes (word).

Function Code

The function code bits indicate the type of bus cycle currently running. Function code information is valid when $\overline{AS}$ is asserted. The function code values are listed in the following table (L = low input or 0, H = high input or 1):

FC2:FC0	Bus Access Type
L L L	reserved
L L H	System/370 space operand
L H L	System/370 space instruction
L H H	reserved
H L L	reserved
H L H	control space operand
H H L	control space instruction
H H H	interrupt acknowledge / service cycle

Function code 7 identifies a Motorola interrupt acknowledge cycle (if A3–A1 ≠ 000) or a sense or control service cycle (if A3–A1 = 000).

Address Strobe

Address bus is valid at high-to-low transition.

Read/Write

Determines direction of data transfer. R/$\overline{W}$ = 1 means Micro/370 is reading data.

Word Strobe/Service Address Bit 3

For write operations, signifies valid data on D31–D0, which the device should store in the addressed word. Micro/370 asserts $\overline{WS}$ only when A1 = 0, guaranteeing data aligned on a word boundary. $\overline{WS}$ does not apply to read operations.

During Micro/370 service cycles, $\overline{WS}$, $\overline{UDS}$, and $\overline{LDS}$ serve as service address bits ($\overline{SA3}$–$\overline{SA1}$). The alternate use of these pins applies to all service cycles other than the interrupt acknowledge cycle. The service address specifies the particular sense or control cycle in progress. For output service cycles (control), the output service strobe ($\overline{VMA}$/$\overline{OSS}$) is used as a data strobe.

Upper Data Strobe/Service Address Bit 2

Signifies use of D31–D24 for transfer of upper byte of addressed halfword on read or write. See the description of $\overline{WS}$/$\overline{SA3}$ for the definition of the alternate use of $\overline{UDS}$ during service cycles.

Lower Data Strobe/Service Address Bit 1

Signifies use of D23–D16 for transfer of lower byte of addressed halfword on read or write operation. See the description of $\overline{WS}$/$\overline{SA3}$ for the definition of the alternate use of $\overline{LDS}$ during service cycles.

Signal Name	Mnemonic	Direction	Output Device Type
Address Bus	A31-A1	Output	Tristate
Real Address Tag	A32	Output	Tristate
Data Bus	D31-D0	Input/Output	Tristate
Function Code	FC2-FC0	Output	Tristate
Address Strobe	$\overline{AS}$	Output	Tristate
Read/Write	R/$\overline{W}$	Output	Tristate
Word Strobe/Service Address Bit 3	$\overline{WS}$/$\overline{SA3}$	Output	Tristate
Upper Data Strobe/Service Address Bit 2	$\overline{UDS}$/$\overline{SA2}$	Output	Tristate
Lower Data Strobe/Service Address Bit 1	$\overline{LDS}$/$\overline{SA1}$	Output	Tristate
Word Transfer Acknowledge	$\overline{WTACK}$	Input	—
Data Transfer Acknowledge	$\overline{DTACK}$	Input	—
Byte Transfer Acknowledge	$\overline{BTACK}$	Input	—
Bus Error	$\overline{BERR}$	Input	—
Data Transfer Exception	$\overline{DTEXN/2}$-$\overline{DTEXN/0}$	Input	—
Bus Request	$\overline{BR}$	Input	—
Bus Grant	$\overline{BG}$	Output	Push-Pull
Bus Grant Acknowledge	$\overline{BGACK}$	Input	—
Interrupt Priority Level	$\overline{IPL2}$-$\overline{IPL0}$	Input	—
Priority Request	$\overline{PR1}$-$\overline{PR0}$	Input	—
System/370 Interruption Requests	$\overline{IR4}$-$\overline{IR0}$	Input	—
Enable	E	Output	Push-Pull
Valid Peripheral Address	$\overline{VPA}$	Input	—
Valid Memory Address/Output Service Strobe	$\overline{VMA}$/$\overline{OSS}$	Output	Tristate
Auxiliary Reset	$\overline{XRESET}$	Input	—
Reset	$\overline{RESET}$	Input/Output	Open Drain
Halt	$\overline{HALT}$	Input/Output	Open Drain
System/370 State Indicators	I/1-I/0	Output	Push-Pull
Input Clock	CLK	Input	—
Bus Cycle in Progress	$\overline{BCIP}$	Output	Push-Pull
Select Read Cycle	3T/$\overline{4T}$	Input	—
Suppress Full Word Write	$\overline{SUPPRESS}$	Input	—
Dual Mode Enable	$\overline{DUAL}$	Input	—
Macrocycle in Progress	$\overline{MIP}$	Output	Push-Pull

Figure E.2 Micro/370 signal summary

Word Transfer Acknowledge
Device response indicating that a full word has been transferred.

Data Transfer Acknowledge
Device output signal indicating bytes signified by $\overline{\text{UDS}}$ and $\overline{\text{LDS}}$ have been transferred.

On a full word write (with $\overline{\text{WS}}$), this device response indicates that only the upper halfword was stored, and Micro/370 will perform a second write operation to transfer the lower halfword on D31–D16.

Byte Transfer Acknowledge
Device output signal indicating byte on D31–D24 has been transferred.

On a full word write (with $\overline{\text{WS}}$), this device response indicates that only the upper byte was stored, and Micro/370 will run more bus cycles to transfer the remainder of the data.

On a halfword write (with $\overline{\text{UDS}}$ and $\overline{\text{LDS}}$), this device response indicates that only the upper byte was stored, and Micro/370 will perform a second write operation to transfer the lower byte on D23–D16.

Bus Error
Device response in bus cycle to terminate current bus cycle without data transfer. Micro/370 response to $\overline{\text{BERR}}$ can be to rerun the current bus cycle if $\overline{\text{HALT}}$ also is asserted.

Data Transfer Exception
Device response in bus cycle to indicate no transfer of data occurred. The bus controller reports the specific exception to the processor.

Bus Request
Input wire ORed with potential bus masters. Indicates another device is requesting bus mastership.

Bus Grant
Output indicates Micro/370 will release control of the bus at the end of the current cycle.

Bus Grant Acknowledge
Input indicates that another device has become bus master.

Interrupt Priority Level
Three input bits signify the Motorola interrupt level.

Priority Request

These inputs request system actions:

$\overline{PR1}$:$\overline{PR0}$	System Action
H H	no action
H L	System/370 Stop
L H	System/370 Start
L L	E-Unit Dump

System/370 Interruption Requests

Activation of the interruption requests causes Micro/370 to perform the standard System/370 interruption response for the following interruptions:

Input	System/370 Interruption Class
$\overline{IR4}$	Restart
$\overline{IR3}$	External
$\overline{IR2}$	Program
$\overline{IR1}$	Machine Check
$\overline{IR0}$	Input/Output

Enable

This signal is the Enable signal used by MC6800-type peripherals. It is a free-running clock with a period of ten Micro/370 clock periods (six clocks low, four clocks high). Enable continues to run regardless of the state of the bus controller. The period of E (1 cycle per 10 Micro/370 cycles) enables interfacing 1-MHz MC6800 peripherals to a 10-MHz Micro/370.

Valid Peripheral Address

Input indicates that the addressed device is an MC6800 family device, and that the data transfer should be synchronized to the Enable signal.

Valid Memory Address/Output Service Strobe

Output indicates that the address bus contains a valid address for MC6800 peripherals. $\overline{VMA}$ is issued in response to a $\overline{VPA}$ input, which indicates that the addressed device was a peripheral.

During Micro/370 output service cycles, the $\overline{VMA}$ pin is used as a data strobe. The alternate name for the pin in this situation is $\overline{OSS}$, and the data tansfer is synchronized to the transition of

this pin. The timing of the signal is similar to that of the data strobes during normal bus output cycles.

Auxiliary Reset

$\overline{\text{XRESET}}$ is used in combination with $\overline{\text{RESET}}$ and $\overline{\text{HALT}}$ to generate external reset requests. $\overline{\text{XRESET}}$ is also used to cancel erroneous device responses during the bus cycle, and to unconditionally rerun bus cycles during a macrocycle (an uninterruptible series of bus cycles).

Reset

As an input, $\overline{\text{RESET}}$ is used in combination with $\overline{\text{XRESET}}$ and $\overline{\text{HALT}}$ to generate external reset requests. $\overline{\text{RESET}}$ can be driven by Micro/370 in response to a Diagnose instruction in order to reset devices external to Micro/370.

Halt

On input, $\overline{\text{HALT}}$ asserted alone has no effect on Micro/370. When $\overline{\text{HALT}}$ is asserted along with a normal acknowledge to a bus cycle ($\overline{\text{WTACK}}$, $\overline{\text{DTACK}}$, or $\overline{\text{BTACK}}$), Micro/370 will not start another bus cycle until $\overline{\text{HALT}}$ is removed. $\overline{\text{HALT}}$ also serves as a modifier to $\overline{\text{BERR}}$ for bus cycle rerun requests. $\overline{\text{HALT}}$ also is used in combination with $\overline{\text{XRESET}}$ and $\overline{\text{RESET}}$ to generate external reset requests.

On output, Micro/370 asserts $\overline{\text{HALT}}$ only when it enters Checkstop state.

System/370 State Indicators

I/1-I/0 indicate the current state of the Micro/370 processor:

I/1:I/0	System/370 State
L L	Load or Operating
L H	Stopped
H L	Checkstop
H H	undefined

Input Clock

The clock input is a TTL-level signal that drives all operations on the Micro/370 chip. The chip runs nominally at 15-MHz, worst case 10-MHz. The clock signal must always run and conform to pulse width specifications.

Bus Cycle in Progress

$\overline{BCIP}$ signifies the start of a new bus cycle one clock cycle earlier than $\overline{AS}$ does.

Select Read Cycle

Low input enables Micro/370 to attempt to complete read bus cycles in three clock cycles (3T) by expecting device responses 1T earlier than required by the Motorola MC68000 bus protocol. This signal is intended to be fixed, either high or low.

Suppress Fullword Write

Low input prevents Micro/370 from attempting fullword write bus cycles. All processor-requested fullword accesses are then performed in parts by the bus controller. This improves performance when no device can respond with $\overline{WTACK}$. This signal is intended to be fixed, either high or low.

Dual Mode Enable

Low input enables Micro/370 Dual mode operation. If $\overline{DUAL}$ is not asserted, all bus cycles will access System/370 memory space. This signal is intended to be fixed, either high or low.

Macrocycle in Progress

Output indicates Micro/370 is running an uninterruptible series of bus cycles. During macrocycles, bus arbitration is suspended and bus cycle rerun requests are disallowed. $\overline{MIP}$ is asserted when Micro/370 performs read-modify-write memory operations or PSW swaps on interruption.

Micro/370 Device Attachment

Through extensions to the Motorola MC68000 bus protocol, Micro/370 allows attachment of a wide variety of devices. Figure E.3 shows interfaces to the different types of devices. Figure E.4 shows a simple system configuration for Micro/370.

The basic Motorola MC68000 component bus protocol supports direct interfacing with 8- and 16-bit memory-mapped devices through the use of the data strobes. Data transfer width selection is done dynamically during the bus cycle by the processor's request for 1 or 2 bytes. Micro/370 bus protocol extends this concept to full word (32-bit) transfers during a single bus cycle. During any Micro/370 bus access, addressed devices may respond by transferring all the requested data, or only a portion of the data. Micro/370 senses these conditions and dynamically

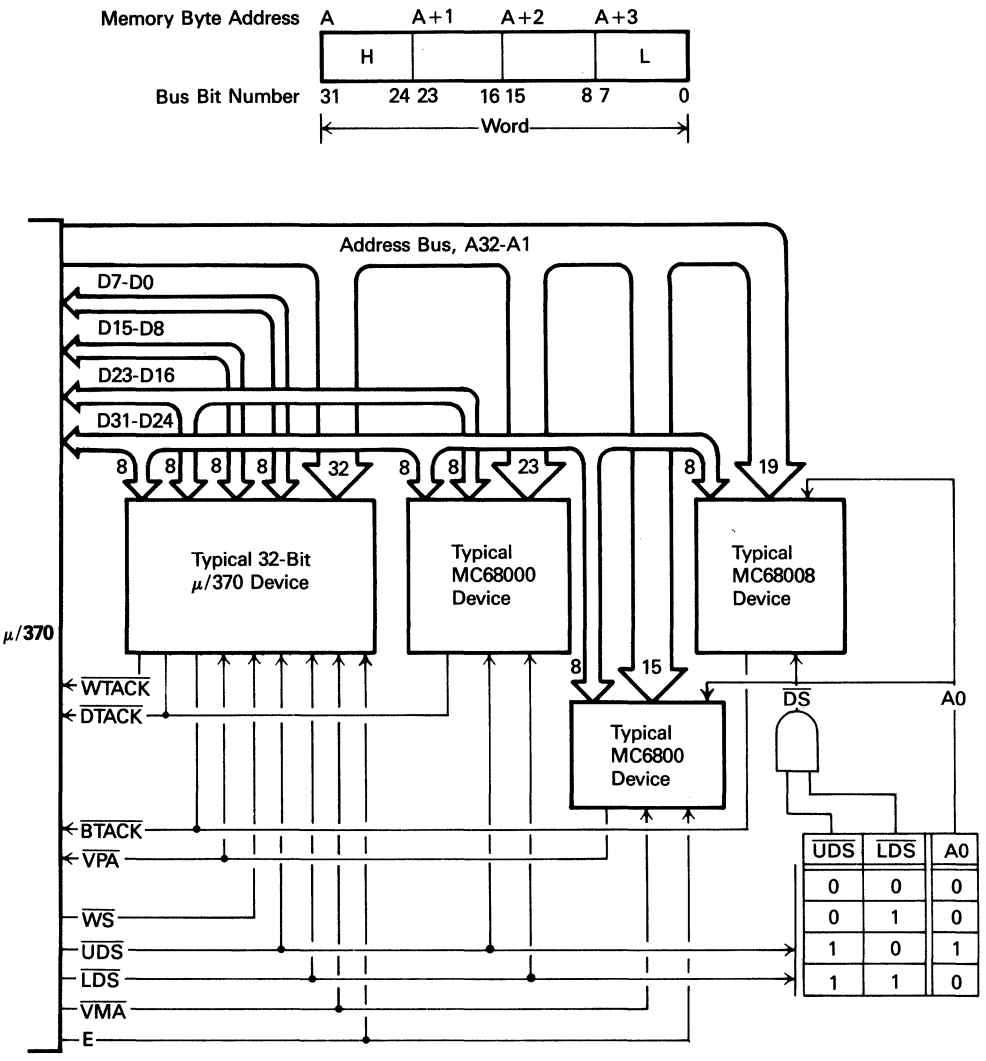


Figure E.3 Micro/370 device attachment

executes additional bus accesses, if required, to complete the storage access.

Devices of varying data widths may therefore be included in a Micro/370 environment with little or no additional user logic, as shown in figure E.3. This holds particularly true for Motorola M68000-family components. Through the use of the $\overline{\text{VMA}}$, $\overline{\text{VPA}}$,

and E pins, the extensive line of MC6800 peripherals is directly compatible with Micro/370.

The full 32-bit addressing capability of Micro/370 allows for a real memory address space of 4 gigabytes. In addition to normal memory access bus cycles a second type of bus cycle, called a service cycle, is available. Service cycles provide a separate address space, redefine address pins, and make full device response status available for software interpretation.

Motorola bus cycles include detecting error responses and honoring requests that bus cycles be rerun. Micro/370 extends the number of data transfer exceptions greatly. These exceptions cover conditions such as storage key checks, address translation exceptions, program exceptions, and machine check conditions.

For interruptions, the MC68000 interface defines a seven-level encoded interrupt scheme. Micro/370 includes this in addition to the System/370 interruptions described in the signal summary. Two other pins are used to initiate System/370 Start and System/370 Stop functions.

The Micro/370 interface includes MC68000 compatible bus arbitration to allow the transfer of control of the bus to alternate bus masters.

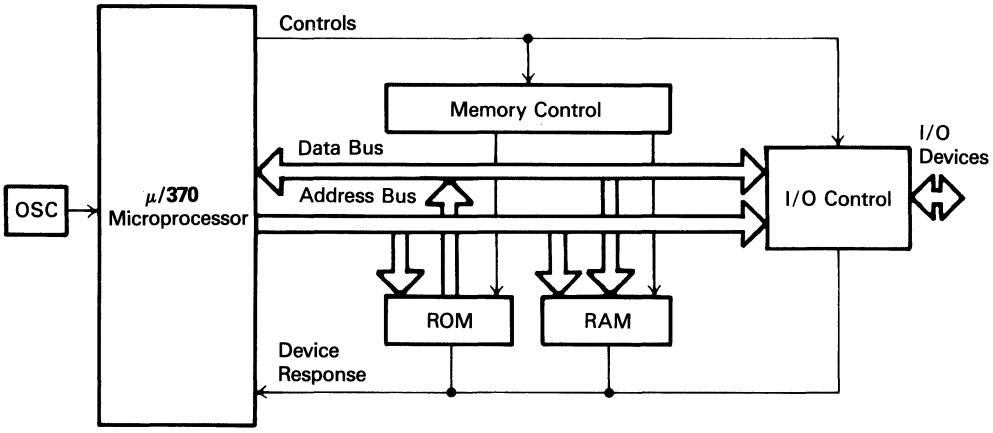


Figure E.4 Simple system configuration for Micro/370

Micro/370 System Design

Micro/370 provides address space separation to ease the design of external hardware. The existence of various address spaces to fulfill specific functions allows partitioning and simplifies decoding logic. Typical address spaces are shown in figure E.5.

The primary means of address space separation is by the function code pins. During every bus cycle, the type of bus access is displayed on these pins. Decoding logic driven by the function code may be used to select the proper address space for the bus operation. The significance of the various address spaces is discussed below.

Two main types of bus cycles occur on the Micro/370 bus. These are memory access cycles and service cycles.

There are two types of memory access cycles, read and write. As the name implies, memory access cycles by Micro/370 access memory (real or virtual). Four address spaces are defined by the function codes. They are System/370 instruction space, System/370 operand space, control instruction space, and control operand space. Normally, programs executed by Micro/370 run in the System/370 spaces. Instruction emulation functions execute from the control spaces (see "Dual Mode Feature" below). Transfer of control between System/370 and control space is automatic under several conditions.

There are three types of service cycles: sense service cycles, control service cycles, and Motorola interrupt acknowledge service cycles. Coprocessors, translation buffers, interrupt memory, and other peripherals reside in the Micro/370 service cycle address space. Micro/370 service cycle bus operations are distinguished by function code 7 ("111" hex) and A3–A1 = 000. Motorola interrupt acknowledge service cycles occur with function code 7 and A3–A1 reflecting the acknowledged Motorola interrupt level code.

Micro/370 operates only in System/370 extended-control (EC) mode. Its System/370 program status word (PSW) cannot take on the basic control (BC) mode format compatible with System/360. Hence PSW bit 12, the EC mode bit, is always set. The Micro/370 PSW (figure E.8, page 319) differs from the System/

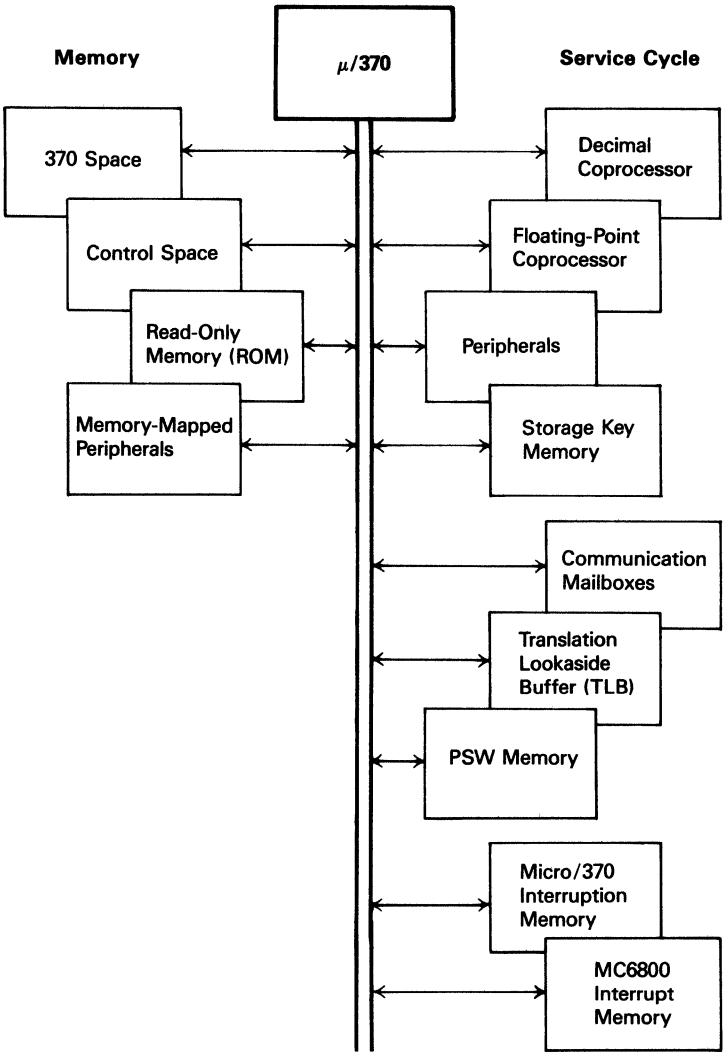


Figure E.5 Typical Micro/370 address spaces

370 EC mode PSW only in having a 32-bit instruction address, in bits 32–63.

Coprocessor Support

Micro/370 does not directly execute the fifty-one floating- and extended floating-point instructions or the nine decimal instructions defined in the System/370 architecture. These instructions can be executed by attached coprocessors. A math coprocessor can be attached to handle the floating-point coprocessing, and a separately attached decimal coprocessor may be used for the decimal instructions. All instruction fetching and decoding is done by Micro/370. When a floating-point or decimal instruction is encountered, Micro/370 determines whether the appropriate coprocessor is there.

If an attached coprocessor for the instruction is found, Micro/370 performs supporting operations for that instruction (including possible operand fetching) and starts the coprocessor with a service cycle. After waiting for the coprocessor to complete its operation, Micro/370 finishes the instruction execution by setting the appropriate System/370 condition codes or performing exception processing. When a required coprocessor is not found, Micro/370 follows the same procedure as for an unimplemented instruction (described below).

Dual Mode Feature

To ease system implementation of System/370 instructions not executed by Micro/370, the Dual mode feature is available. Dual mode consists of logic to support control space, sixteen additional 32-bit general-purpose registers, and the Diagnose Dual Mode instruction. System/370 instructions residing in control space are used to emulate other System/370 instructions; control space can be considered a "writable control store." With several restrictions, instructions fetched from control space may operate on data in either control or 370 space.

Pulling the $\overline{\text{DUAL}}$ pin low causes Micro/370 to operate with Dual mode on. The processor then switches between two modes, Control mode and 370 mode, and accesses instructions and operands in two spaces, control space and 370 space. With Dual mode on, sixteen additional general-purpose registers—called shadow registers—can be accessed by a program running in Control mode. Programs running in 370 mode read the general registers and simultaneously write general registers and the corresponding shadow registers. This feature lowers the data movement overhead necessary when a Control mode program

gets control, since the shadow registers already have all data presently in the general registers. The Diagnose Dual Mode instruction switches modes and controls access to spaces, register sets, and mode indicators.

Virtual Storage Support

Micro/370 supports System/370 virtual storage in the following ways. (Virtual storage system implementation with Micro/370 is, however, optional.) Micro/370 does operand pretesting to avoid page-fault conditions arising during instruction execution. Trial execution is done for the Translate (TR) instruction. The real address tag (A32) indicates whether the address appearing on the address pins is real or virtual. External user logic manages the translation of virtual addresses to real storage addresses. Exception conditions (page translation, segment translation, and translation specification) arising from address translation operations are recognized by Micro/370. These are encoded on Micro/370's data transfer exception lines by external logic.

Diagnose Instructions

System/370 architecture defines the Diagnose instruction as model dependent. Micro/370 includes in its instruction set fourteen Diagnose suboperations to aid system designers. With the Diagnose instruction set, the Micro/370 programmer can do the following:

370 mode Diagnose suboperations:

- Set or store the Micro/370 Motorola 68000 interrupt mask
- Store the current System/370 PSW
- Add two fullword, unsigned, packed decimal numbers
- Subtract two fullword, unsigned, packed decimal numbers
- Calculate effective address(es) of an instruction's operand(s)
- Directly control 63 Micro/370 pins (with service cycles)
- Sense pending System/370 and MC68000 interruptions
- Assert the MC68000 Reset pin (Reset out operation)
- Dump Micro/370 execution unit contents to memory

Control mode Diagnose suboperations:

- Select general or shadow registers for reading and writing
- Select operand spaces (control or 370 space) for fetching and storing
- Store current mode indicators
- Load current System/370 PSW and switch to 370 mode
- Turn on instruction tracing

Micro/370 Processor States

Figure E.6 shows the Micro/370 processor states and the transitions among the states. System/370 control functions are initiated by different combinations of asserted input signals to Micro/370. The first group of pins for this purpose is $\overline{\text{XRESET}}$, $\overline{\text{HALT}}$, and $\overline{\text{RESET}}$. Actions taken for combinations of these inputs are listed in figure E.7.

System/370 CPU control is provided for in the priority request inputs, $\overline{\text{PR1}}$ and $\overline{\text{PR0}}$. A description of these pins is given under

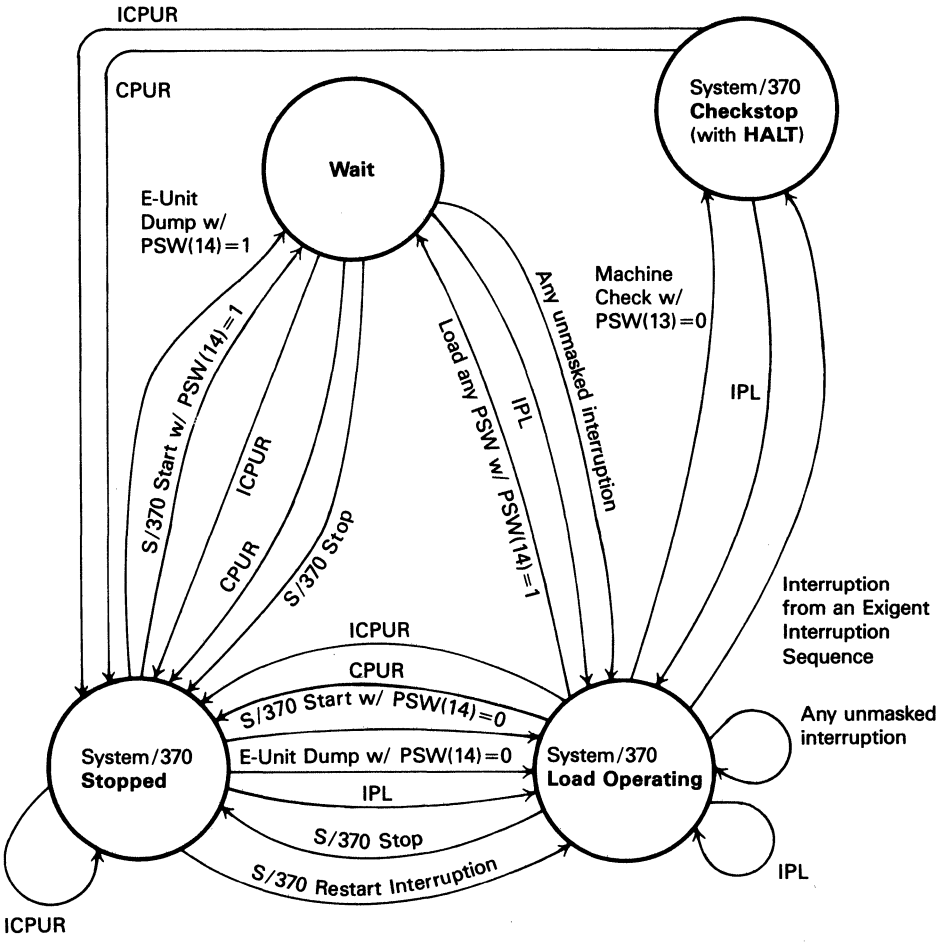


Figure E.6 Micro/370 processor states

"Micro/370 Signal Descriptions." Other interruption actions are initiated with the System/370 interruption requests ($\overline{IR4}$–$\overline{IR0}$) and the interrupt priority level ($\overline{IPL2}$–$\overline{IPL0}$) pins. The priority of Micro/370 response to these inputs is (in descending order):

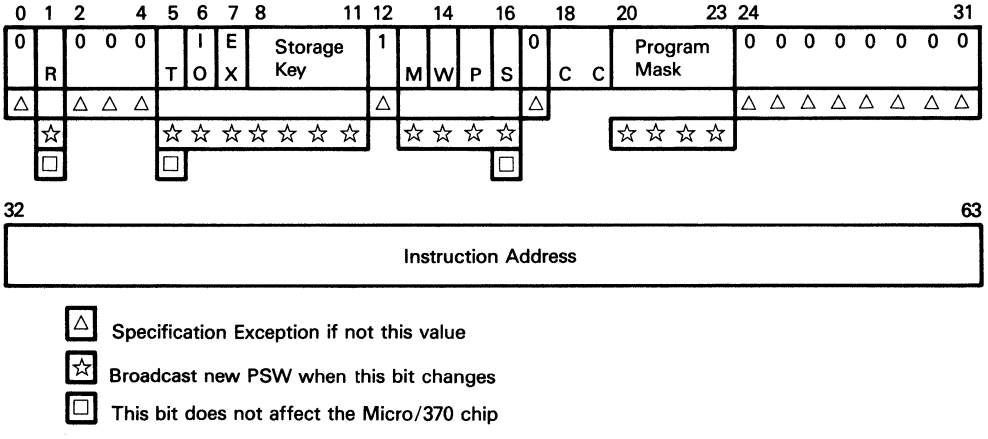
- IPL, CPUR, ICPUR
- Exceptional bus cycle termination
- Trace
- E-Unit dump; System/370 start

(continued next page)

$\overline{XRESET}$	$\overline{HALT}$	$\overline{RESET}$	Action
1	1	1	No operation
1	1	0	No operation[1]
1	0	1	No operation
1	0	0	System/370 initial CPU reset (ICPUR)[1]
0	1	1	No operation[2 3]
0	1	0	No operation[1 2]
0	0	1	System/370 CPU reset (CPUR)
0	0	0	Initial Program Load (IPL)[1 4]

Notes:
1. $\overline{RESET}$ will unconditionally reset external MC68000 peripherals.
2. Reserved for future use.
3. $\overline{XRESET}$ also used for control purposes.
4. Implicitly includes ICPUR.

Figure E.7 Micro/370 resets



△ Specification Exception if not this value

☆ Broadcast new PSW when this bit changes

□ This bit does not affect the Micro/370 chip

Figure E.8 Micro/370 program status word

- Program interruption ($\overline{IR2}$)
- Machine check interruption ($\overline{IR1}$)
- External interruption ($\overline{IR3}$)
- I/O interruption ($\overline{IR0}$)
- Motorola interruption ($\overline{IPL2}$–$\overline{IPL0}$)
- Restart interruption ($\overline{IR4}$)
- System/370 stop
- Prefetch error
- Instruction execution

Comparing System/370 to Micro/370: Total Number of Instructions by System/370 Facility				
Instruction Group			**System/370**	**Micro/370**
Universal Instruction Set			157	91
	System/370	**Micro/370**		
Commercial Instruction Set	113	91		
Standard Instruction Set	104	91		
General Instructions	87	85		
Control Instructions	9	6		
I/O Instructions	8	0		
Decimal Instruction Set	9	0		
Floating-Point Facility	44	0		
Advanced Control Program Support Feature			4	4
	System/370	**Micro/370**		
Conditional Swapping Facility	2	2		
PSW-Key Handling Facility	2	2		
Branch and Save Facility			2	2
Channel-Set Switching Facility			2	0
CPU Timer and (TOD) Clock Comparator Facility			4	0
Direct Control Facility			2	0
Dual-Address Space (DAS) Facility			12	0
Extended Facility: IPTE and TPROT instructions			2	0
Extended-Precision Floating-Point Facility			7	0
Move Inverse Facility			1	1
Multiprocessing (MP) Facility			4	0
Recovery Extensions Facility			1	0
Storage-Key-Instruction Extensions Facility			3	0
Suspend and Resume Facility			1	0
Test Block Facility			1	0
Translation Facility			5	4
		Total	**209**	**102**

Figure E.9 Micro/370 instruction summary

Mnemonic	Instruction Name	Mnemonic	Instruction Name
1. A	Add	52. MVCL	Move Long
2. AH	Add Halfword	53. MVI	Move
3. AL	Add Logical	54. MVN	Move Numerics
4. ALR	Add Logical	55. MVO	Move with Offset
5. AR	Add	56. MVZ	Move Zones
6. BAL	Branch and Link	57. N	AND
7. BALR	Branch and Link	58. NC	AND
8. BAS	Branch and Save	59. NI	AND
9. BASR	Branch and Save	60. NR	AND
10. BC	Branch on Condition	61. O	OR
11. BCR	Branch on Condition	62. OC	OR
12. BCT	Branch on Count	63. OI	OR
13. BCTR	Branch on Count	64. OR	OR
14. BXH	Branch on Index High	65. PACK	Pack
15. BXLE	Branch on Index Low or Equal	66. PTLB	Purge TLB
16. C	Compare	67. RRB	Reset Reference Bit
17. CDS	Compare Double and Swap	68. S	Subtract
18. CH	Compare Halfword	69. SH	Subtract Halfword
19. CL	Compare Logical	70. SL	Subtract Logical
20. CLC	Compare Logical	71. SLA	Shift Left Single
21. CLCL	Compare Logical Long	72. SLDA	Shift Left Double
22. CLI	Compare Logical	73. SLDL	Shift Left Double Logical
23. CLM	Compare Logical Characters under Mask	74. SLL	Shift Left Single Logical
24. CLR	Compare Logical	75. SLR	Subtract Logical
25. CR	Compare	76. SPKA	Set PSW Key from Address
26. CS	Compare and Swap	77. SPM	Set Program Mask
27. CVB	Convert to Binary	78. SR	Subtract
28. CVD	Convert to Decimal	79. SRA	Shift Right Single
29. D	Divide	80. SRDA	Shift Right Double
30. DIAG	Diagnose	81. SRDL	Shift Right Double Logical
31. DR	Divide	82. SRL	Shift Right Single Logical
32. EX	Execute	83. SSK	Set Storage Key
33. IC	Insert Character	84. SSM	Set System Mask
34. ICM	Insert Character under Mask	85. ST	Store
35. IPK	Insert PSW Key	86. STC	Store Character
36. ISK	Insert Storage Key	87. STCM	Store Characters under Mask
37. L	Load	88. STH	Store Halfword
38. LA	Load Address	89. STIDP	Store CPU ID
39. LCR	Load Complement	90. STM	Store Multiple
40. LH	Load Halfword	91. STNSM	Store Then AND System Mask
41. LM	Load Multiple	92. STOSM	Store Then OR System Mask
42. LNR	Load Negative	93. SVC	Supervisor Call
43. LPR	Load Positive	94. TM	Test under Mask
44. LPSW	Load PSW	95. TR	Translate'
45. LR	Load	96. TRT	Translate and Test
46. LTR	Load and Test	97. TS	Test and Set
47. M	Multiply	98. UNPK	Unpack
48. MH	Multiply Halfword	99. X	Exclusive OR
49. MR	Multiply	100. XC	Exclusive OR
50. MVC	Move	101. XI	Exclusive OR
51. MVCIN	Move Inverse	102. XR	Exclusive OR

Figure E.10 Micro/370 instruction list (by mnemonic)

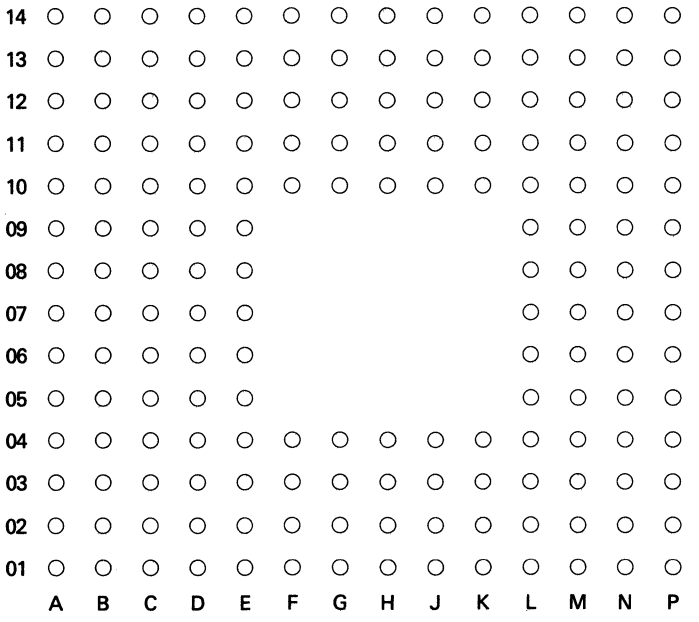

14 O O O O O O O O O O O O O O
13 O O O O O O O O O O O O O O
12 O O O O O O O O O O O O O O
11 O O O O O O O O O O O O O O
10 O O O O O O O O O O O O O O
09 O O O O O O O O
08 O O O O O O O O
07 O O O O O O O O
06 O O O O O O O O
05 O O O O O O O O
04 O O O O O O O O O O O O O O
03 O O O O O O O O O O O O O O
02 O O O O O O O O O O O O O O
01 O O O O O O O O O O O O O O
A B C D E F G H J K L M N P

Figure E.11 Micro/370 package pinout (bottom view)

Micro/370 Execution Unit

The execution unit, or "E-Unit," is the main data path for the microprocessor. A block diagram of the E-Unit appears in figure E.14 (page 325). Descriptions of the E-Unit components are presented here to show how Micro/370 performs data operations.

External Buses—E-Unit communication with the chip address and data pads occurs through the address bus control (ABC) and data bus control (DBC) components.

ABC—Address bus control conditionally increments addresses on nonword-boundary memory stores.

DBC—Data bus control moves data into the correct byte(s) of DI from the pads, or moves data from the correct byte(s) of DO to the pads.

The ABC and DBC are controlled by the bus controller, shown in figure E.13.

Internal Buses—Two 32-bit internal data buses (named A and B) span the E-Unit. They are each split, by bidirectional bus couplers, into two sections (P and D). These buses provide paths for up to four concurrent data transfers, in a single processor cycle. The P and D sections may be coupled, when necessary, to

Pin Number	Signal Name	Pin Number	Signal Name	Pin Number	Signal Name	Pin Number	Signal Name
A01	D16	D02	D18	G13	TP1[2]	M01	D7
A02	D25	D03	D27	G14	TPP[2]	M02	D6
A03	D29	D04	SCANIN4[1]	H01	D22	M03	D5
A04	D30	D05	VDD	H02	VDD	M04	D4
A05	E	D06	$\overline{\text{XRESET}}$	H03	GND	M05	VDD
A06	$\overline{\text{IR1}}$	D07	VDD	H04	GND	M06	VDD
A07	$\overline{\text{IR2}}$	D08	VDD	H10	GND	M07	GND
A08	$\overline{\text{IR3}}$	D09	$\overline{\text{BGACK}}$	H11	VDD	M08	VDD
A09	$\overline{\text{IR4}}$	D10	$\overline{\text{DTEXN/2}}$	H12	GND	M09	VDD
A10	A32	D11	$\overline{\text{DTEXN/1}}$	H13	VDD	M10	A25
A11	I/1	D12	$\overline{\text{WTACK}}$	H14	CLK	M11	A26
A12	FC2	D13	$\overline{\text{BTACK}}$	J01	$\overline{\text{IPL2}}$	M12	A27
A13	$\overline{\text{AS}}$	D14	SCANOUT2[1]	J02	D9	M13	A28
A14	$\overline{\text{HALT}}$	E01	D11	J03	VDD	M14	A24
B01	D15	E02	D19	J04	GND	N01	D3
B02	D24	E03	VDD	J10	VDD	N02	D1
B03	D28	E04	ITEST[1]	J11	VDD	N03	A1
B04	D31	E05	GND	J12	$\overline{\text{TPROC}}$[1]	N04	A3
B05	$\overline{\text{PR0}}$	E06	VDD	J13	TCRESET[1]	N05	A2
B06	$\overline{\text{DUAL}}$	E07	GND	J14	CBTEST[1]	N06	A5
B07	I/0	E08	GND	K01	$\overline{\text{IR0}}$	N07	VDD
B08	R/$\overline{\text{W}}$	E09	$\overline{\text{DTEXN/0}}$	K02	$\overline{\text{IPL1}}$	N08	A16
B09	$\overline{\text{LDS}}$/$\overline{\text{SA1}}$	E10	GND	K03	$\overline{\text{PR1}}$	N09	A17
B10	$\overline{\text{UDS}}$/$\overline{\text{SA2}}$	E11	SCANIN2[1]	K04	D23	N10	A18
B11	FC1	E12	SCANOUT1[1]	K10	SCAN[1]	N11	A19
B12	$\overline{\text{VMA}}$/$\overline{\text{OSS}}$	E13	$\overline{\text{TCLK}}$[2]	K11	GND	N12	A20
B13	$\overline{\text{BR}}$	E14	TCLK[2]	K12	TRISTATE[1]	N13	A22
B14	$\overline{\text{DTACK}}$	F01	D12	K13	LSSDB[1]	N14	A23
C01	D14	F02	D20	K14	LSSDA[1]	P01	D2
C02	D17	F03	VDD	L01	D8	P02	D0
C03	D26	F04	GND	L02	SCANIN3[1]	P03	A4
C04	SCANOUT4[1]	F10	VSX[2]	L03	$\overline{\text{MIP}}$	P04	A6
C05	$\overline{\text{IPL0}}$	F11	VDD	L04	SCANOUT3[1]	P05	A7
C06	3T/$\overline{\text{4T}}$	F12	TP4[2]	L05	GND	P06	A8
C07	$\overline{\text{WS}}$/$\overline{\text{SA3}}$	F13	TPR[2]	L06	GND	P07	A9
C08	FC0	F14	TP3[2]	L07	GND	P08	A10
C09	$\overline{\text{BCIP}}$	G01	D13	L08	GND	P09	A11
C10	$\overline{\text{BG}}$	G02	D21	L09	GND	P10	A12
C11	$\overline{\text{RESET}}$	G03	VDD	L10	CTEST[1]	P11	A13
C12	$\overline{\text{SUPPRESS}}$	G04	GND	L11	A30	P12	A14
C13	$\overline{\text{BERR}}$	G10	TP2[2]	L12	A31	P13	A15
C14	$\overline{\text{VPA}}$	G11	GND	L13	A29	P14	A21
D01	D10	G12	EUTEST[1]	L14	SCANIN1[1]		

Notes:
1. Connect to GND during normal operation.
2. No connection during normal operation.

Figure E.12 Micro/370 pin assignments

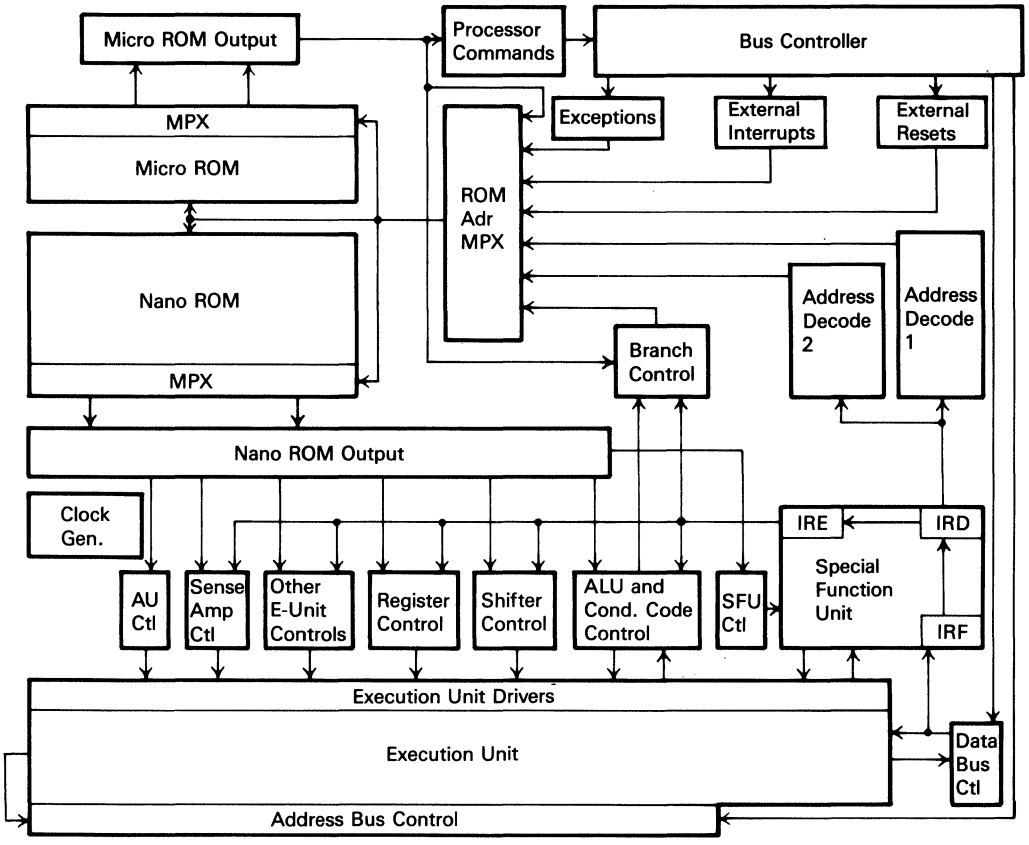


Figure E.13 Micro/370 chip block diagram

transfer data between components residing on different sections of the A and B buses.

The internal buses have sense amplifiers to eliminate the need for large bus drivers in each E-Unit component. They sense when a source is attempting to drive the bus and amplify the signal to drive as many as 3 receivers. Separate sense amplifiers exist for the P and D sections, for use when the internal buses are operating uncoupled. When coupled, the D sense amplifiers drive the entire bus.

A second function is performed by the D sense amplifiers. They are arranged in bytes and can be separately selected for data transfer. When a sense amplifier is not turned on, no data transfer occurs. This mechanism allows loading of individual bytes of the various registers.

Micro/370 Special Function Unit

The special function unit has its own 32-bit bus (called C). The C bus sets to 0 any bits not driven by a source.

AO—The 32-bit Address Out buffer is at the left end of the E-Unit. It supplies addresses to address bus control (ABC) for external bus accesses.

EX—Micro/370 uses the 32-bit EX register for processing the System/370 Execute instruction. The EX register maintains a pointer to the Execute instruction.

PSWL—This register is related to the low-order 32-bit half of the System/370 PSW. Unlike the standard System/370 PSW, which has only a 24-bit instruction address, the entire 32 bits of PSWL are an address.

PC—The 32-bit PC register is the progam counter for Micro/370. When Micro/370 prefetches the next instruction, it updates PC.

AU—The arithmetic unit is a 32-bit adder (it also performs subtraction). Micro/370 uses AU for address calculation at the same

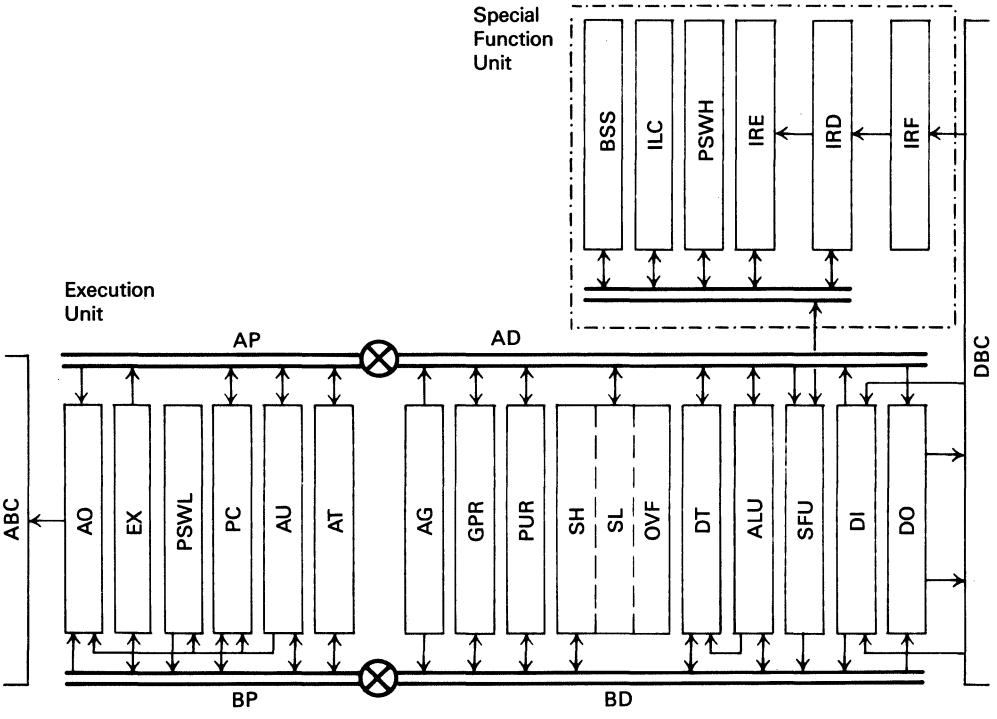


Figure E.14 Micro/370 execution unit and special function unit

time some other operation is being performed by the ALU. AU contains the address of the next halfword in the instruction stream if no branch is taken.

AT—The 32-bit Address Temporary register is used as a temporary register during operations, especially for address calculations.

AG—The address generator generates 32-bit constants. It can provide data for either the A or B bus. Many of the constants available are fixed addresses in System/370 architecture, for swapping PSWs or for storing interruption codes.

GPR—Thirty-two 32-bit registers. Sixteen of these are the System/370 general registers. The other sixteen are the shadow registers.

Either the general registers or the shadow registers may be selected as data sources in E-Unit data transfers. Both may be selected as destinations for transfers.

PUR—Micro/370 uses the 8-bit Pack-Unpack Register for instructions that manipulate 4-bit nibbles of data. Even though PUR stores only 1 byte of data, it generates a complete 32-bit word when it is the source for a data transfer.

SH/SL (Shifter High/Shifter Low)—The Micro/370 shifter can shift 64 bits of data 0 to 63 bits in either direction in one processor cycle. The shifter can operate on the entire 64 bits or on the high-order word alone. An overflow unit concurrently detects arithmetic left-shift overflow conditions.

DT—Data Temporary is a 32-bit register also used for temporary values.

ALU—A variety of arithmetic and logic functions is provided by the Micro/370 arithmetic and logic unit. Under processor control, various addition, subtraction, incrementing, logical AND, logical OR, and logical XOR operations can be performed on A bus and B bus contents.

The ALU generates several carry-out bits, including a decimal-arithmetic correction-factor based on the carry out of each 4-bit nibble. Seven condition signals are generated for carry, negative, overflow, and zero flags for the full 32-bit result as well as the low-order byte.

DI—The 32-bit Data Input register receives data from off-chip via data bus control (DBC).

DO—The 32-bit Data Output register sends data off-chip via data bus control (DBC).

SFU—The 32-bit SFU register links the special function unit to the E-Unit.

BSS—Bus Status Selection is an 8-bit data path between the E-Unit and various parts of the microprocessor.

ILC—A register in the special function unit that is related to the 2-bit System/370 instruction length code. Its value represents the length of the current System/370 instruction in halfwords.

PSWH—Micro/370 maintains the high-order half of the System/370 PSW in PSWH. It is a 32-bit register in the special function unit.

IRF, IRD, and IRE—Micro/370 uses three halfword registers to prefetch instructions: the instruction register for fetch (IRF), instruction register for decode (IRD), and instruction register for execution (IRE). They reside in the special function unit. During instruction execution, IRD and IRE each contain the first halfword of the instruction, and IRF contains the following halfword.

Annotated Bibliography

Hadsell, R. W., "Micro/370," *Microarchitecture of VLSI Computers,* NATO ASI Series E, No. 96, Martinus Hijhoff, Dodrecht, The Netherlands, 1985, pp. 3–54. Details of the internal logic of the Micro/370 chip.

Ong, S., Chao, H. H., Tsai, M. Y., Shih, F.-H. W., Hou, J. C. L., Lewis, K. W., Tang, J. Y.-F., Trempel, C. A., Hadsell, R. W., Yu, H. N., McCormick, P. E., Davis Jr., C. V., Diamond, A. L., Medve, T. J., and Higham, J. P., "A 32b Single-Chip Microprocessor," *ISSCC Digest of Technical Papers,* February 1986, pp. 28–29, 291. Formal announcement of the chip (as a research project) to the technical community.

Tredennick, Nick, "How to Flowchart for Hardware," *Computer,* December 1981, pp. 87–102. Tutorial on the design method used to design the logic of the Micro/370 chip, written by one of the main designers of Micro/370.

Greier, P. F., Franch, R. L., Shih, F. W., Hadsell, R. W., Tang, J. Y. F., Ong, S., and Chao, H. H., "Micro/370 Functional Test," Research Report, RC 11728, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, February 18, 1986.

Chao, H. H., Ong, S., Tsai, M. Y., Shih, F.-H. W., Hou, J. C. L., Lewis, K. W., Tang, J. Y.-F., Trempel, C. A., Hadsell, R. W., Yu, H. N., Greier, P. F., Franch, R. L., McCormick, P. E., Davis Jr., C. V., Diamond, A. L., and Medve, T. J. 19896. "Micro/370, a 32-bit single-chip microprocessor." The Journal of Solid State Circuits, Vol. SC-21, No. 5 (October), pp. 733–740.

APPENDIX

F

A One-Semester Design Project

This text is organized to support a concurrent semester-long microprocessor design project. Students, working in pairs, can be expected to do flowcharts and logic design for a simplified processor. Here's how the project might be organized.

Before the semester begins, the instructor should select a processor architecture and simplify it (if necessary). Prepare a user's manual for the (simplified) processor. Give the user's manual to the students at the first lecture. I taught a computer design course in the spring of 1984 at the University of California–Berkeley. The students did a project based on a subset of the instructions from the MC68000. They worked directly from the *MC68000 16/32-Bit Microprocessor Programmer's Reference Manual* (fourth edition). I handed out a list (see figure F.1) showing which instructions, address modes, and operand lengths to implement. Students worked directly from the user's manual. Students also should be given documents showing drawing standards and other rules for the course.

The first two chapters of this book introduce the topic and define a microprocessor. The first lectures should cover these chapters and the architecture of the assigned processor. Useful homework during this period is to have students write simple

XX	ABCD	Add Decimal with Extend
	ADD	Add Binary
	ADDA	Add Address
	ADDI	Add Immediate
XX	ADDQ	Add Quick
XX	ADDX	Add Extended
	AND	AND Logical
	ANDI	AND Immediate
	ANDI to CCR	AND Immediate to Condition Codes
	ANDI to SR	AND Immediate to Status Register (P)
	ASL	Arithmetic Shift Left
	ASR	Arithmetic Shift Right
	Bcc	Branch Conditionally
	BCHG	Test a Bit and Change
	BCLR	Test a Bit and Clear
	BRA	Branch Always
	BSET	Test a Bit and Set
	BSR	Branch to Subroutine
	BTST	Test a Bit
XX	CHK	Check Register against Bounds
XX	CLR	Clear an Operand
	CMP	Compare
	CMPA	Compare Address
	CMPI	Compare Immediate
	CMPM	Compare Memory
	DBcc	Test Condition, Decrement, and Branch
	DIVS	Signed Divide
	DIVU	Unsigned Divide
	EOR	Exclusive OR Logical
	EORI	Exclusive OR Immediate
	EORI to CCR	Exclusive OR Immediate to Condition Codes
	EORI to SR	Exclusive OR Immediate to the Status Register (P)
	EXG	Exchange Registers
	EXT	Sign Extend
	ILLEGAL	Illegal Instruction
	JMP	Jump
	JSR	Jump to Subroutine
	LEA	Load Effective Address
	LINK	Link and Allocate
	LSL	Logical Shift Left
	LSR	Logical Shift Right
	MOVE	Move Data from Source to Destination
	MOVE to CCR	Move to Condition Codes
	MOVE to SR	Move to the Status Register (P)
	MOVE USP	Move User Stack Pointer (P)
	MOVEA	Move Address
	MOVEM	Move Multiple Registers
XX	MOVEP	Move Peripheral Data
XX	MOVEQ	Move Quick
XX	MULS	Signed Multiply
XX	MULU	Unsigned Multiply
XX	NBCD	Negate Decimal with Extend
XX	NEG	Negate
XX	NEGX	Negate with Extend
	NOP	No Operation
XX	NOT	Logical Complement
	OR	Inclusive OR Logical
	ORI	Inclusive OR Immediate
	ORI to CC	Inclusive OR Immediate to Condition Codes
	ORI to SR	Inclusive OR Immediate to the Status Register (P)
	PEA	Push Effective Address
	RESET	Reset External Devices (P)
XX	ROL	Rotate (without Extend) Left
XX	ROR	Rotate (without Extend) Right
XX	ROXL	Rotate with Extend Left
XX	ROXR	Rotate with Extend Right
XX	RTE	Return from Exception (P)
XX	RTR	Return and Restore Condition Codes
	RTS	Return from Subroutine
XX	SBCD	Subtract Decimal with Extend
	Scc	Set According to Condition
XX	STOP	Load Status Register and Stop (P)
	SUB	Subtract Binary
	SUBA	Subtract Address
	SUBI	Subtract Immediate
XX	SUBQ	Subtract Quick
XX	SUBX	Subtract with Extend
XX	SWAP	Swap Register Halves
	TAS	Test and Set an Operand
XX	TRAP	Trap
XX	TRAPV	Trap on Overflow
	TST	Test an Operand
	UNLK	Unlink

Do not implement instructions marked XX.
Implement only byte and long operations.
Implement all address modes.

Figure F.1 Processor instruction set

assembly language programs with reference to the user's manual for the experimental processor. The assembly language programming homework helps students become familiar with the architecture of the processor.

Chapter 3 introduces flowcharts, which are fundamental to the entire design procedure. They should be covered thoroughly in lecture. Students should be asked to submit homework flowcharts for the processor address modes (or some similarly reduced subset of the architecture). These flowcharts should be audited carefully to make sure that the students know what they are doing before launching into the major work of the project. Short interviews with the design groups can be beneficial for the students and informative for the instructor. Students could be asked, for example, to show preliminary work, their design objectives, and a schedule for their design project.

Once the students understand the material in chapter 3, work on the project can begin in earnest. Students can begin working on flowcharts for all the instructions and address modes in their user's manual. Their progress should be monitored periodically.

Example homework shown later in this appendix is taken from a computer design course I taught at the University of California–Berkeley in the spring of 1984.

Sample Semester Homework and Lecture Plan

Week 1

- Hand out (simplified) processor user's manual.
- Hand out drawing standards and other rules for the course.
- Hand out homework assignment #1.
- Lecture chapter 1: Overview.
- Lecture chapter 2: What is a microprocessor?

Week 2

- Hand out homework assignment #2.
- Lecture chapter 3: Flowcharts: introduction, prerequisites, overview.
- Lecture chapter 3: Flowcharts: objectives and notation.
- Collect homework assignment #1.

Week 3

- Hand out homework assignment #3.
- Lecture chapter 3: Level 1 flowcharts.
- Lecture chapter 3: Level 1 flowcharts (continued).
- Collect homework assignment #2.

Week 4

- Hand out homework assignment #4.
- Lecture chapter 3: Level 2 flowcharts.
- Lecture chapter 3: Level 2 flowcharts (continued), review.
- Collect homework assignment #3.

Week 5

- Lecture chapter 4: Controller overview, flowchart-hardware relationship.
- Lecture chapter 4: Implementation: execution unit, decoders, control word.
- Collect notes from homework assignment #4.
- Oral review of project planning (scheduled separately from class).

Week 6

- Hand out homework assignment #5.
- Lecture chapter 4: Example implementation: PC, T2, register control examples.
- Lecture chapter 4: Example implementation: ALU control, state sequencer.
- Collect homework assignment #4.

Week 7

- Midterm exam.
- Lecture chapter 5: Overview, clocking, timing.
- Collect notes from homework assignment #5.

Week 8

- Hand out homework assignment #6.
- Lecture chapter 5: Exceptions.
- Lecture chapter 5: Control store address selection.
- Collect homework assignment #5.

Week 9

- Hand out homework assignment #7.
- Lecture chapter 5: Control store, control word decoders.

- Lecture chapter 5: Internal communication, microcode, bus interface, mode control.
- Collect homework assignment #6.

Week 10

- Hand out homework assignment #8.
- Lecture chapter 6: Operation overview, execution overlap, prefetching.
- Lecture chapter 6: Clocking and timing, bus sense amps, shifter control.
- Collect homework assignment #7.

Week 11

- Lecture chapter 7: Implementation sequence, example instructions.
- Lecture chapter 7: Example instructions (continued).
- Collect homework assignment #8.

Week 12

- Lecture chapter 7: Conversion instructions, multiply, divide.
- Lecture chapter 8: Implementation sequence, address out, Pack-Unpack control.
- Oral review of project (scheduled separately from class)

Week 13

- Lecture chapter 8: Special function unit example.
- Lecture chapter 9: Chip design methods.
- Project formal documents due.

Week 14

- Lecture Appendix A: Cultures of microprogramming.
- Lecture: Review, discuss final exam.
- Oral presentation of project (scheduled separately from class).

Week 15

- Final Exam

Homework	15%	Project	35%
Midterm exam	25%	Final exam	25%

Suggested Lecture Topics by Chapter

Chapter 1: Here's the Deal

- Overview

Chapter 2: Defining a Microprocessor

- What is a microprocessor?

Chapter 3: Hardware Flowcharts

- Introduction, prerequisites, and overview
- Objectives and notation
- Level 1 flowcharts
- Level 1 flowcharts (continued)
- Level 2 flowcharts
- Level 2 flowcharts (continued) and review

Chapter 4: Implementing from Flowcharts

- Controller overview and relationship between flowcharts and hardware
- Sample implementation: execution unit, instruction decoders, and control word format
- Sample implementation: PC, T2, and register control examples
- Sample implementation: ALU control example and state sequencer

Chapter 5: How a Microprocessor Works

- Overview, clocking, and timing
- Exceptions
- Control store address selection
- Control store and control word decoders
- Internal communication, microcode, bus interface, and mode control

Chapter 6: The IBM Micro/370 Microprocessor

- Operation overview, execution overlap, and prefetching
- Clocking and timing, bus sense amps, and shifter control

Chapter 7: Hardware Flowcharts for Micro/370

- Implementation sequence, example instructions
- Example instructions (continued)
- Conversion instructions, multiply, and divide

Chapter 8: Implementing Micro/370 from Flowcharts

- Implementation sequence, address out, and Pack-Unpack control
- Special function unit example

Chapter 9: VLSI Design Method(ologie)s

- Methods A, B, and C

Appendix A: The "Cultures" of Microprogramming

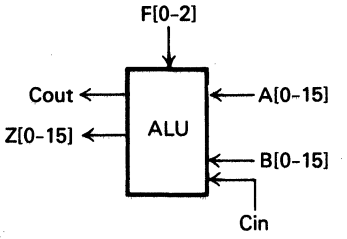
- Cultures of microprogramming and contrasting folklore with reality

Problem Set 1

Due _____

1. Design a 16-bit arithmetic and logic unit (ALU) with the functions outlined in figures F.1 and F.2. Use only NAND gates. First design a single-bit "slice" of the ALU, then interconnect sixteen of them to form a full 16-bit ALU.
2. What is the worst case delay time for a single operation in your ALU? Describe the path and calculate the delay.
3. What is the cost in gates?

Your grade will be based on correctness, speed (depth of the worst case propagation path for a 1-bit slice), and cost in gates.



F[0-2]	3-bit ALU function control
A[0-15]	16-bit input operand
B[0-15]	16-bit input operand
Z[0-15]	16-bit result
Cin	1-bit carry in
Cout	1-bit carry out

ALU Control Table

F[0-2]	Operation	Description
000	ADD	Add A and B
001	ADDC	Add A and B and Cin
010	SUB	Subtract B from A
011	SUBC	Subtract B from A with Borrow
100	AND	AND of A and B
101	OR	OR of A and B
110	ARS	Arithmetic Right Shift
111	ALS	Arithmetic Left Shift

Figure F.2 ALU block diagram and control table

Problem Set 2
Due _____

This homework is designed to familiarize you with the instruction set of the processor architecture.

All programs should contain copious comments.

1. Write a program to perform a bubble sort on byte data beginning at location 2000 hex in memory. Assume that general register 1 holds the number, *n*, of items to be sorted.
2. Implement a circular queue of length 128 words—that is, write two subroutines, one to insert data and one to extract data from the queue.
3. An algorithm for performing division of unsigned integers is as follows: Put the value 1 into a marker word and shift it and the divisor left until the divisor is greater than the dividend. Shift the marker and divisor right one place at a time, and in cases where the shifted divisor is less than the dividend, subtract it from the dividend and add the marker to the quotient. When sufficient right shifts have moved the 1 bit out of the marker, the division is finished and the manipulated dividend now holds the remainder.

Write a program to perform division using the above method.

Problem Set 3
Due _____

1. Do trial flowcharts for the following instructions:

NOP	
ADD.L Dn,Dm	ADD register direct long words
ADD.L Dn,d[An,Xi]	ADD address register indirect with index
BRA	branch always
MOVE.B Abs.L,d[An]	move byte

2. Propose an execution unit for your design.

Read Chapter 3: Hardware Flowcharts. Your flowcharts should look similar to those in figure 3.12.

I suggest a bus-oriented design for the execution unit. Your drawing of the execution unit should look similar to the drawing of figure 3.4 but with more detail.

Do the trial flowcharts by assuming you can do any transfers you want. See what buses you need. More buses cost more money, but if there are too few, performance is worse. Begin building the execution unit with the elements you know you must have, including the programmer's register set, the PC, an ALU, the status register, and an instruction register. Then do the trial flowcharts and add the buses and elements you need to accomplish the tasks efficiently. This is an engineering design problem. You decide what you need in the execution unit to do the job best. You will not have to use the same execution unit for the rest of the project. It should evolve into just what you need by the end of the semester. If it is not what you want, change it.

Problem Set 4
Notes due ____
Due _____

1. Find a partner and decide on the execution unit you will begin with.
2. Do level 1 flowcharts for all the address modes.
3. Do level 1 flowcharts for all the instructions in the first column of the processor instruction set table (Figure F.1). This should be ADD through MOVE to SR.

Read chapter 3: Hardware Flowcharts. Your flowcharts should look similar to those in figure 3.11. If the execution unit does not work correctly, change it. It should evolve with the level of detail in the design.

Split this assignment with your partner. Turn in the notes as a single assignment and turn in the final results as a single assignment. Do not do flowcharts for instructions with word operands in registers or data memory. You need not do ADD.W, but you should do BRA.W, for example.

Turn in a copy of the work you have done by _____. This need not be the form for the final assignment. Show at least the execution unit you are using and the level 1 flowcharts for all the address modes. The complete assignment is due on _____(one week later).

If you want a copy of your homework, copy it before you turn it in (you may not get the original back).

Problem Set 5
Notes due ____
Due _____

1. Do level 2 flowcharts for all the address modes and for all the instructions.

Read chapter 3: Hardware Flowcharts. Your flowcharts should look similar to those in figure 3.12. If the execution unit does not work correctly, change it. It should evolve with the level of detail in the design.

Split this assignment with your partner. Turn in the notes as a single assignment and turn in the final results as a single assignment. Do not do flowcharts for instructions with word operands in registers or data memory. You need not do ADD.W, but you should do BRA.W, for example.

Turn in a copy of the work you have done by _____. This need not be the form for the final assignment. Show at least the execution unit you are using and the level 2 flowcharts for half of the instructions. The complete assignment is due on _____(one week later).

If you want a copy of your homework, copy it before you turn it in (you may not get the original back).

Problem Set 6
Due _____

1. Write the following special microsequences:
   a. Reset
   b. Interrupt
   c. Illegal and Unimplemented Instructions
   d. Privilege Violations
   e. Bus Error
   f. Address Error
2. Derive your microword format. One way to do this is to tabulate all the uses of each of the execution unit and controller pieces that the microword controls. Then assign a field in the microword with enough bits to decode to the necessary control lines.

3. Show your controller block diagram. The diagram should include the execution unit (as a block), the branch control logic (as a block, no detail), the instruction decoder(s) (as a block or blocks), the control store, the control store address sources, the control store address multiplexers, the bus controller (as a block), and the interrupt logic (as a block).

The *MC68000 16/32-Bit Microprocessor Programmer's Reference Manual* (fourth edition) contains a description of the processing required for the various special microsequences required. It also tells how the conditions are recognized. Read Section 4: Exception Processing, particularly pages 40–47.

The complete assignment is due _____ . If you want a copy of your homework, copy it before you turn it in (you may not get the original back).

Problem Set 7
Due _____

1. Tabulate the branch conditions used in your flowcharts. Show an implementation for your branch control unit.
2. Show your ALU and Condition Code Control Table and show an implementation for your table. Include selection and saving of the appropriate condition codes and the equations you implemented for the various condition code settings.
3. Derive and show your instruction decoder(s). Show how you handle the Unimplemented and Illegal instructions and Privilege Violations.

The complete assignment is due _____ . If you want a copy of your homework, copy it before you turn it in (you may not get the original back).

Problem Set 8
Due _____

1. Show the block diagram and the logic design for your bus controller. Include the handshaking logic, bus arbitration, I/O data multiplexers, and interrupt logic. The interrupt logic includes detection, synchronization, ranking, and control store address generation (to point to the special interrupt microsequence).

2. Complete a performance chart using your flowcharts.

   Directions for the performance chart: Use the instruction frequencies listed in figure F.3 as the horizontal axis of your chart. List the instructions on the horizontal axis in order of

Frequency	Instruction	Frequency	Instruction
19%	Bcc.W	1%	ADDA.W d(An,Xi),An
10	MOVE.L d(An,Xi),Dn	1	BRA.W
5	MOVEA.L d(An,Xi),An	1	SUB.W d(An,Xi),Dn
4.9	LEA d(An,Xi),An	1	AND.W d(An,Xi),Dn
4.1	TST.B d(An,Xi)	1	MOVE d(An,Xi),SR
43%		97%	
4	Bcc.B	.5	RTS
3.9	MOVE.L Dn,d(An,Xi)	.5	NOP
3.2	MOVE.L Dn,Dn	.4	EXG Dn,Dn
3	MOVE.B d(An,Xi),d(An,Xi)	.4	ADDI.B d(An,Xi)
2.9	CMPI.B d(An,Xi)	.2	EORI.B d(An,Xi)
60%		99%	
2.8	TST.L Dn	.2	SUBI.B d(An,Xi)
2.5	SUB.L Dn,Dn	.2	EOR.W d(An,Xi),Dn
2.4	CMPM.B (Ay)+,(Ax)+	.1	OR.W d(An,Xi),Dn
2.3	ADD.L Dn,Dn	.1	ASL.W #8,Dy
2	BSR.W	.1	ASR.W #8,Dy
72%		99.7%	
1.8	LINK An,#d	.05	LSR.W #8,Dy
1.7	MOVEM.L d(An,Xi),#FFFF	.05	SUBA.W d(An,Xi),An
1.6	CMP.L Dn,Dn	.04	UNLK An
1.5	CMP.L d(An,Xi),Dn	.04	TAS (An)
1.4	JSR d(An,Xi)	.02	EXT.L Dn
80%		99.9%	
1.4	JMP d(An,Xi)	.01	BCHG.B Dn,Abs.L
1.4	MOVE.B Imm,d(An,Xi)	.01	BCLR.B Dn,Abs.W
1.4	MOVE.B Dn,d(An,Xi)	.01	BSET.L Dn,d(PC)
1.4	MOVEM.L #FFFF,d(An,Xi)	.01	BTST.L Dn,d(PC,Xi)
1.4	DBcc	.01	MOVE d(An,Xi),CCR
87%		99.95%	
1.3	LSL.W #8,Dy	.01	MOVE SR,-(An)
1.3	CMPA.L An,An	.01	MOVE USP,An
1.3	ORI.B d(An,Xi)	.01	PEA d(PC)
1.1	ANDI.B d(An,Xi)	.01	Scc -(An)
1	ADD.W d(An,Xi),Dn	.01	RESET
92%		100.0%	

Notes:
Bcc branches successfully 60% of the time.
DBcc never branches on the condition code.
DBcc branches on count 80% of the time.

Figure F.3 MC68152 instruction mix

frequency of occurrence, beginning with Bcc. Draw the chart on the same paper you are using for the flowcharts. There will be four vertical axes.

1. Plot cumulative frequency of occurrence using the letter F.
2. Plot cumulative microwords using the letter M. Count each word only once. Normalize the scale to 100 percent of the words in your control store. Your plot should rise slowly to something less than 100 percent (since a bunch of microwords are used in interrupt and reset microsequences you don't get political credit for).
3. Plot the cumulative average cycles per instruction using the letter μ. Use a separate scale for the values (do not normalize). For each instruction, compute the average cycles per instruction for all instructions up to that point, where f_i is the frequency of the ith instruction and c_i is the number of microcycles it takes to do the ith instruction.

$$\mu_n = \frac{\sum_{i=1}^{n} f_i c_i}{\sum_{i=1}^{n} f_i}$$

4. Plot the cumulative average bus utilization using the letter B. Use a separate scale for the values (do not normalize). For each instruction, compute the average cycles per instruction for all instructions up to that point, where b_i is the number of bus cycles it takes to do the ith instruction.

$$B_n = \frac{\sum_{i=1}^{n} \frac{f_i b_i}{c_i}}{\sum_{i=1}^{n} f_i}$$

The complete assignment is due _____ . If you want a copy of your homework, copy it before you turn it in (you may not get the original back).

A P P E N D I X

G

Glossary

This glossary of technical terms is explanatory, not definitive, as in "Here's what I mean."

Active Logic 1; true.

Active high Describes a signal where the higher voltage denotes the logic 1 state. Also referred to as positive logic.

Active low Describes a signal where the lower voltage denotes the logic 1 state. Also referred to as negative logic.

Actual (transistors) Transistors that are actually implemented. In (silicon gate) MOS technology, you potentially create a transistor each time poly crosses diffusion. These potential transistors are called transistor sites. You get an actual transistor only when poly crosses diffusion over a thin oxide (as opposed to a thick oxide—both are silicon dioxide).

Address mode Information in an instruction that says how to calculate the address of the operand(s). IBM refers to address modes as "forms of an instruction."

AG (Micro/370) The address generator. This unit sits above the execution unit but has a 32-bit path connecting it to the A or B bus. AG is the source for special constants and addresses (such as reserved storage locations in System/370) required by Micro/370.

ALU (Micro/370) The 32-bit arithmetic and logic unit. The ALU is used for all general-purpose arithmetic. It can write to the A or

B bus or directly to the DT register. One input to the ALU always comes from the B bus, and the other comes either from a constant generator or the A bus. The ALU contains a result register.

AO (Micro/370) The 32-bit address output buffer. It contains the drivers for the address pads. AO gets the addresses from either the A or B bus or directly from the AU.

APW Automatic placement and wiring.

Architect (*a*) The author of the architecture; (*b*) nonexistent verb.

Architecture An English-language specification. It is the view of the computer seen by an assembly language programmer. IBM calls it the "principles of operation." Others call it the "user's manual" or "programmer's reference manual."

Asynchronous (Hardware) Not synchronized with the system clock. An event is asynchronous if its occurrence depends on signals other than the system clock. It is like saying, "We'll leave for lunch when she gets to my office," as opposed to leaving at noon sharp, no matter what. (Compare *Synchronous*.)
(Software) Not synchronized with the execution of instructions. An event is asynchronous if instruction execution can proceed independently of the event's occurrence.

AT (Micro/370) The 32-bit Address Temporary register connected to the A and B buses.

AU (Micro/370) The 32-bit arithmetic unit. The arithmetic unit, which does only two's complement addition and subtraction, is used primarily for address calculation and instruction address updates. Its output is connected (optionally) to either the A or B bus, AO, PSWL, and PC. One input to the AU comes from the A bus. The other input can come from either the B bus or a constant generator. The AU contains a result register.

Automatic placement and wiring Having a computer program specify where groups of circuits should be on a chip and how they should be wired together.

BC mode (IBM System/370) basic-control mode; System/360 compatibility under System/370.

Bipolar technology A semiconductor technology using bipolar transistors.

Bipolar transistor Transistor turned on or off by current.

Bit numbering (Motorola) Right-most (least significant) bit is bit 0.
(IBM) Left-most (most significant) bit is bit 0.

Block diagram A drawing, consisting primarily of labeled boxes, showing the logical structure of part of a computer. A block diagram with more than four boxes and two buses is at the limits of comprehension by management.

Buffer (Hardware) A latch with output drivers; (software) temporary main storage used for input/output.

Buffer storage (IBM) cache.

Bus A collection of signal lines.

Bus controller The logic on a microprocessor chip controlling the protocol at the chip's pins and providing synchronized signals to the state sequencer.

Bus coupler A FET device acting like a relay on a data bus. It lets the bus be split into two buses dynamically.

Bus cycle The time it takes to run a complete transaction on the external bus (a memory read or write, for example). In Micro/370, a bus cycle takes a minimum of 3T or 4T (read) or 4T (write). The actual time depends on the speed of the memory system.

Byte 8 bits; the minimum addressable quantity in a memory.

Cache A smaller, faster memory that sits between the CPU and the main memory to give the main memory the appearance of higher speed. CPUs are faster than memory, so caches can serve a speed-matching function. In microprocessor-based systems, caches can lessen the traffic on the too-busy CPU-memory bus. (The CPU-memory bus is the performance bottleneck in microprocessor-based systems.) The benefit of caches is, in practice, largely unproven and almost always overstated in microprocessor-based systems.

CAD Computer-aided design.

CAE Computer-aided engineering.

Capacitor A device that behaves like a balloon that can be inflated by electrons. (It is harder to inflate as the balloon begins to get full.) Capacitors can, therefore, absorb electronic "shock" (spikes) in a signal.

CC Condition code. Also, command mnemonic to invoke the C compiler in UNIX.

Central processing unit The engine that runs a computer.

Channel The diffusion area under the gate in a FET.

Chip Rectangular slice of a semiconductor wafer containing electronic circuits. Most chips are made using a combination of photographic and chemical processes. It turns out that if you pile

layers of silicon on top of each other, each layer containing subtle differences at the atomic level, you can make areas that act like transistors, other areas that act like capacitors, and other areas that act like resistors.

Chip recipe (aka "technology," "fabrication process"):

- Spread goop on silicon wafer.
- Spread light-sensitive goop on top.
- Shine light through a photographic negative. (Makes certain places acid-resistant—in the pattern of the negative.)
- Dump in acid. (Creates the pattern by selective eating away.)
- Rinse.
- Bake in oven at high heat. (This step puts in the "subtle" differences above.)
- Repeat with new negative.

  The "negatives" above are called "masks." It takes ten or more of these masks. A chip recipe is about two hundred steps and can take weeks, even months to complete. Chip recipes are trade secrets.

Circuit Electronics word for "facility."

Circuit count Vague. The question is "What's a circuit?" Sometimes people talk in terms of "equivalent circuits" (such as two-input NANDs). There is no general agreement on the amount of logic incorporated in "one circuit."

Circuit design The part of chip design concerned with power dissipation, propagation delays, voltage level, rise times, loading, device sizes, and other electrical properties.

CISC Pejorative term for computers not categorized as RISCs. (Originally, complex instruction set computer.)

Clock An oscillator. A signal (usually a square wave) used to synchronize transfers between states.

Clock phase A periodic pulse related to the clock.

Clock speed Oscillation rate.

CMOS Complementary metal oxide semiconductor; a semiconductor technology using nMOS and pMOS transistor pairs so that no currents are allowed to flow except during the time when the transistors are opening or closing their paths.

Combinational logic Synonym for combinatorial logic, hardwired logic, random logic.

Combinatorial logic Synonym for combinational logic, hardwired logic, random logic.

Computer-aided design A misnomer for computer programs that effectively interfere with design procedures. "Computer-restricted design" is apt.

Computer-aided engineering Computer-aided design.

Condition code An attribute of the result of an ALU operation. I identify four condition codes as part of what I call the standard set. They are:

- C—carry out (this is borrow-not for subtraction)
- N—sign of the result (1 is negative, 0 is positive)
- V—overflow (significant result bits lost)
- Z—zero result

Control line A wire that propagates the signal to the gate of a transistor, which, in turn, blocks or allows another signal transfer.

Control logic Logic that transforms source signals to control lines.

Control sequence A sequence of control words or flowchart states.

Control store On-chip read-only memory (ROM) holding microcode.

Control word The bit pattern you get from the control store in response to an input address.

CPU Central processing unit.

Cycle A unit of processing time.

- Clock cycle—a unit of clock operation (one 0–1 transition and one 1–0 transition). Designated T.
- CPU cycle—the time it takes to do a unit of processing in the CPU. How much processing one "unit" is depends on how fancy the execution unit is. The time to execute a single flowchart state. In Micro/370, a CPU cycle takes 2T.
- Microcycle—CPU cycle in a microcoded implementation. Emphasizes that in a microcoded implementation, a CPU cycle is the time to execute a single microcode control word. Precisely, the number of clock cycles between control store accesses.
- Bus cycle—the time it takes to run a complete transaction on the external bus (such as a memory read or write). In Micro/370, a bus cycle takes a minimum of 3T or 4T (read) or 4T (write). The actual time depends on the speed of the memory system.

DA Design automation.

DAT (IBM System/370) Dynamic address translation. IBM term for virtual memory addressing capability.

Data flow Execution unit.

Data path(s) Execution unit.

Decoder Logic that maps an (input) bit pattern to some new (output) bit pattern.

Design The conversion of one level of formal description into another. For example, the conversion of the English-language specification into the logic specification.

Design automation The idea of replacing engineers with machines. It is believed this can be done by people who understand what engineers *should* do, not what engineers *actually* do. Also called DA.

Destination The place to which a signal is sent, such as the ALU.

Device MOS transistor, resistor, or capacitor.

DI (Micro/370) The 32-bit Data Input register. Its output can be connected to the A or B bus.

Diffusion A pure silicon area doped (atomically impregnated) with an impurity (boron or phosphorous) to make it a semiconductor.

DO (Micro/370) The 32-bit Data Output register. Its input can come from the A or B bus.

Don't care (Digital logic) A combination of input variables that cannot occur. Since you know the input combination cannot occur, you "don't care" what value is assigned to the output. This helps minimize logic.

Dotted AND Several output lines joined so that any logic 0 (zero) will make the set value a logic 0.

Dotted OR Several output lines joined so that any logic 1 will make the set value a logic 1.

Doubleword 64 bits. In memory, a doubleword is located on a doubleword (multiple of 8) boundary.

Driver An amplifier.

DT (Micro/370) A 32-bit Data Temporary register connected to the A and B internal buses. The ALU can write directly to DT.

Dynamic (Of circuits) Yielding valid output values only for a little while, even though the input remains unchanged. (Compare *Static*.)

ECL Emitter coupled logic; differential signal, current-mode logic. It is very fast, high power, and bipolar. Example logic family is Motorola's MECL 10K.

EC mode (IBM System/370) extended-control mode.

Edge triggered Occurrence of an event depends on a rising (or falling) signal transition. You do something based on whether you saw a signal go from low to high, for example. The static level of the signal does not matter. Similar to a nontypematic key on a keyboard. (You don't care if the signal is logic 1 or 0, only that it changed from, say, 0 to 1.) (Compare *Level triggered.*)

Effective address The CPU's idea of where to find the operand. The instruction format may define operand addresses in terms of a base register and a displacement. The CPU adds the displacement to the contents of the base register to generate the "effective address." (In this case, the effective address is the address the CPU sends to the memory system to access the operand.)

Enable A line controlling the propagation or storage of a value.

E-unit Execution unit.

EX (Micro/370) The Execute register. This register remembers the original instruction address during an Execute instruction. It has a read path from the A bus and a bidirectional path to the B bus.

Execution unit Also called "data flow" or "data paths." That part of Micro/370 containing the data flow: the registers, the ALUs, the special function units, and the internal data buses.

Fabrication (Chips) Manufacturing.

Facility Synonym for "thing." Has zero information content. (It could be used, for example, after every entry facility in this glossary facility.)

FET Field-effect transistor.

FET technology A semiconductor technology using field-effect transistors.

Field-effect transistor Transistor turned on or off by voltage.

Floor plan A drawing showing where the major physical macros are located on the chip.

Flowcharts A graphic representation of state sequencer operation. Flowcharts are derived from the English-language

specification. They are used as specification for the microcode and the logic design. Not related to software flowcharts.

Gate (*a*) A Boolean logic element (such as AND, OR, and NAND) made up of transistors. (*b*) The part of a field-effect transistor to which applying a voltage causes charge carriers (electrons or holes) to flow, or not flow, in the channel.

Gate array Gates (such as NAND) placed in an array on a chip, like bottles in a case of beer. You wire them together (usually using a computer program) to achieve combinational logic functions.

General registers (IBM System/370) The sixteen 32-bit general registers.

Global wiring Wires connecting the chip macros to each other.

Halfword (IBM System/370) 16 bits. In memory, a halfword is located on an even address boundary.

Hardwired logic Synonym for combinational logic, combinatorial logic, and random logic.

Housekeeping tasks (Flowcharts) Things the CPU does that do not contribute to the instruction operation but are necessary to continue processing. In executing an ADD instruction, for example, addition of the operands is the instruction operation and everything else is housekeeping. Housekeeping includes program counter update, instruction accesses, and instruction decoding. (Compare *Operation tasks*.)

IC Integrated circuit; chip.

ICU (Micro/370) Interface control unit, name for the Micro/370 bus controller.

IEEE Institute of Electrical and Electronics Engineers.

ILC (Micro/370) The 2-bit instruction length count. ILC holds the length of the instruction being executed. It is above the execution unit and connected via the internal C bus to the SFU.

Implementation The logical representation of a functional specification.

Inactive logic 0; false.

Instruction An instruction defined in the architecture.

Integrated circuit A chip; literally, a "circuit in one place." IBM uses "monolithic," meaning "single-stone."

Integration Putting in one place. The number of transistors on a chip is referred to as the "scale of integration"—the degree of being in one place. Expressed as small, medium, large, very large, and (recently) ultra large scale integration.

Interrupt A signal that disrupts the normal sequence of events. I classify interrupts according to how they affect CPU processing: external, internal, and software.

- External interrupts come from outside the microprocessor. There are two types: immediate external interrupts indicate that some serious problem has occurred and the current bus transaction fails; deferred external interrupts indicate that some device (keyboard or printer, for example) wants service from the CPU.
- Internal Interrupts result from a fault in the current instruction (such as an illegal op code, arithmetic overflow, or divide fault).
- Software interrupts are instructions to the CPU, but to the programmer, it might look like an operating system trap (IBM System/370 SVC instruction, for example).

Interruption (IBM) Interrupt.

IRD (Micro/370) The 16-bit instruction register for decode. Holds the first halfword of the next instruction so decode can begin during execution of the current instruction. It is above the execution unit and connected via the internal C bus to the SFU.

IRE (Micro/370) The 16-bit instruction register for execution. Holds the first halfword of the instruction being executed. It is above the execution unit and connected via the internal C bus to the SFU.

IRF (Micro/370) The 16-bit instruction register for fetching. Usually holds the second halfword following the beginning of the current instruction. It is above the execution unit and connected via the internal C bus to the SFU.

Karnaugh map A graphic representation of a logic function. The representation has the property that adjacent values differ in only a single bit position. Used to minimize implementation logic. This used to be important to keep the component count low. I use Karnaugh maps to minimize PLA logic. I place the control terms in the Karnaugh map and move them around to achieve a minimum PLA implementation. The resulting Karnaugh map assigns the bit patterns for the corresponding microword control

field. (This is useful, but opposite what is normally taught. Usually, you start with fixed map assignments and derive minimum logic to implement the function.)

KIPs One thousand instructions per second (1 KIP is 1,000 instructions per second, not 1,024.)

K-map Karnaugh map.

Layout The geometrical representation of a chip's physical design.

Level triggered Occurrence of an event depends only on the logic level (0 or 1) of a signal and ignores signal transitions. Similar to a typematic key on a keyboard. (Compare *Edge triggered.*)

Load Electrical equivalent of mass. The bigger the mass (load) you carry around in your backpack, the slower you move. The bigger the load on a signal, the slower the signal changes.

Logic (*a*) Description of a design in terms of Boolean values; (*b*) referring to an IC that calculates or decides, as distinct from a memory IC.

Logic 0 false

Logic 1 true

Logic design A procedure that converts a specification into a specification usable by circuit designers.

Logic term The Boolean expression for a single signal.

LSI large scale integration. Describing an IC logic chip containing about one thousand transistors.

Machine CPU

Macro (IBM) A functional unit of physical design such as a register or ALU.

Macro placement The position on a chip of a physical macro.

Mask Glass photographic negative used during chip manufacturing.

Master image A design method based on selecting, placing, and wiring together previously designed, off-the-shelf, physical macros.

Master slice Gate array (chip).

Memory (IBM) storage

Metal Aluminum wiring on a chip. Chips typically have one or two levels of metal (1986).

Methodology Strictly, the theory of methods. Common use, pretentious synonym for "method."

Microcode The contents of a control store. Microcode is the collective term for the bit patterns controlling the actions of the execution unit and the state sequencer. To some people, microcode looks like software; to others, microcode looks like logic.

Microcoded Adjective used with "controller" to indicate dominant means of state sequencing as microcode.

Microcomputer A chip composed of a microprocessor, ROM, RAM, and I/O controllers. Examples: Intel 8048 and Motorola MC6801.

Microcycle CPU cycle in a microcoded implementation. Emphasizes that in a microcoded implementation, a CPU cycle is the time to execute a single microcode control word. Precisely, the number of clock cycles between control store accesses.

Microprocessor A single-chip CPU with at least the following: clock-phase generators, bus controller, control store, state sequencer, instruction decoders, and execution unit. Examples: Intel iAPX286, Motorola MC68020, Zilog Z8000.

Microprogram Microcode.

Microprogrammed Microcoded.

Microword The part of the control store word used to control the state sequencer. In Micro/370, it is 18 bits wide and pops out of the upper control store. Synonym for control (store) word.

MIPs Million instructions per second. (One MIP is 1,000,000 instructions per second, not 1,048,576.) Thought to be a uniform measure of computer performance, like miles per hour. That this is not even close to being true has both enhanced and detracted from many a career.

Module (IBM) IC package.

MOS Metal oxide semiconductor. Synonym for field-effect transistor.

MOSFET Metal oxide semiconductor field-effect transistor (MOS, for short).

MPX Multiplexer.

MSI Medium scale integration. Describing an IC logic chip containing about five hundred transistors.

Multiplexer Logic that selects a single output from several inputs. Like having multiple lines of people converging on a single-chair ski lift and having a person select riders from the lines.

MUX Multiplexer.

MX (Micro/370) A 32-bit multiplexer between what it thinks is the external data bus and both the DI and DO registers.

Nanoword That part of the control word used to control the execution unit. In Micro/370, it is 72 bits wide and pops out the bottom of the lower control store.

Nibble The left or right 4 bits of a byte.

nMOS N-channel MOS. MOS technology created by doping the channel with an n-type impurity (from column V in the Periodic Table of Elements). The majority carriers are electrons.

OCD Off-chip driver.

Op code Operation code; the bits in the instruction format that tell what the operation is.

Operation tasks (Flowcharts) Things the CPU does that contribute to the instruction operation but are not necessary to continue processing. In executing an ADD instruction, for example, addition of the operands is an operation task and everything else is housekeeping. (Compare *Housekeeping tasks*.)

Package The container for a chip(s). IBM term: module.

Pack-Unpack register (PUR) (Micro/370) A special-purpose unit used to aid execution of instructions that move nibbles around. (MVN, MVZ, MVO, PACK, and UNPK are examples.) PUR is connected to the A and B buses, but what happens when an operand arrives at PUR depends on both the bus it comes from and the macro instruction that sent the operand. PUR includes an 8-bit register.

Pads Square areas of metal (usually around the edge of the chip) that are the interface between on-chip and off-chip signals.

Part Chip.

Pass gate An FET used as a relay.

Pass transistor An FET used as a relay.

PC (Micro/370) 32-bit program counter that holds instruction addresses. It has bidirectional connections to the A and B internal buses and a write-only connection from the AU.

PER (IBM System/370) Program-event recording.

Pin A connection to a chip at the package level.

PLA Programmable logic array. Normally, input lines go to a regular AND array; outputs of the AND array go to an OR array; the lines out of the OR array are the output. I use the term to refer to logic structures that "look" normal (even if they do not have AND and OR arrays).

pMOS P-channel MOS. MOS technology created by doping the channel with a p-type impurity (from column III of the Periodic Table of Elements). The majority carriers are holes (not electrons).

Poly Poly(crystalline)silicon. Poly conducts but not as well as metal.

Precharge nMOS circuits pull voltages down much more quickly than they can pull them up. If there is time between uses, circuits and buses often are pulled high (precharged) so a value can be quickly obtained by pulling lines low.

Prefetch An instruction fetch for an instruction other than the one the CPU is currently executing.

Principles of operation (IBM) The document that specifies the architecture of a computer. The architecture known as IBM System/370 is described in the IBM publication *IBM System/370 Principles of Operation*, form no. GA22-7000.

Prioritize Nonexistent verb for "rank."

Processor Central processing unit.

PSW Program status word. Eight bytes of information in a System/370 CPU which define the state of the CPU.

PSWH (Micro/370) The high-order word of the program status word. Above the execution unit and connected via the internal C bus to the SFU.

PSWL (Micro/370) The low-order word of the program status word. It can take data from the A bus and send data to the B bus. It is automatically loaded from the AU between instructions.

Pull-down An FET used as a resistor to pull the voltage toward zero. They are much easier to make than real resistors.

Pull-up An FET used as a resistor to pull the voltage high.

PUR (Micro/370) The Pack-Unpack Register, a special-purpose unit used to aid execution of instructions that move nibbles around. (MVN, MVZ, MVO, PACK, and UNPK are examples.) PUR is connected to the A and B buses, but what happens when an operand arrives at PUR depends on both the bus it comes from and the macro instruction that sent the operand. PUR includes an 8-bit register.

R0–RF (Micro/370) The sixteen 32-bit System/370 general registers.

Race condition Logic permitting signal changes to propagate through more than a single memory element during one clock phase. The final state depends on rate of propagation of the sig-

nals. If, for example, logic on the output of a latch can determine the next state of the latch, a race condition exists. Suppose the latch changes state. The new latch output signal might propagate through the logic in time to influence the next state (before the end of the clock phase enabling the state change), or it might not, depending on the speed of the signal.

RAM Random access memory.

Random logic Pejorative term for "the other person's logic." I use this term to mean logic implemented directly from Boolean equations. Logic consisting only of primitive gates (such as NAND, NOR, NOT, AND, and OR) and not using PLAs or ROMs. No effort is made to force structure into the logic specification for either timing or physical topology. Synonyms: combinational logic, combinatorial logic, hardwired logic.

Realization The physical representation of an architecture. In this case, the silicon chip.

Register Separately addressed (from memory) storage locations, at least conceptually, in the CPU.

Relay A bidirectional current switch controlled by logic. It can be either "open" (no current flow) or "closed" (current flow).

Relay logic Digital logic assuming ideal bidirectional switching elements. Transistors performing relay logic are called "pass transistors" or "pass gates."

RISC Acronym connoting "goodness in hardware, software, and architecture." (Originally, reduced instruction set computer.)

ROM Read-only memory.

ROS Read-only storage; IBM acronym for ROM.

S0–SF (Micro/370) The sixteen 32-bit shadow registers. They are a copy of the System/370 general registers.

SA (Micro/370) Sense amplifiers. There is a set for each section of the A and B buses. The individual bytes of the sense amps for the data sections of each bus can be controlled with a mask.

Semiconductor A material that is sometimes an insulator and sometimes a conductor.

Sense amp An amplifier driven by a differential comparator. In Micro/370, internal A and B buses carry true and complement signals. The buses are initially precharged high, the source signal is then connected to both buses. The sense amp decides which way the signal is being driven and forces it quickly to the desired value. (So the macros connected to a bus can have wimpy drivers—saving area and power.)

Sequence (Flowcharts) A physically contiguous series of states.

SFU (Micro/370) Special function unit. I use it to do all the special things the normal execution unit pieces cannot do. It does nibble swaps (inside a byte) for the instruction. It checks legal signs and digits for CVB. It assembles the funny program counter and mask combination for the BAL and BALR instructions. It is the link between the state sequencer and the execution unit (for access to the instruction registers, the external bus status, the program status word—all internal state registers in the state sequencer). It is represented in the execution unit as a 32-bit register that writes to the B bus and reads from the A bus. (I even use it for temporary storage.) It is connected to many parts of the state sequencer by the 32-bit C bus. The flowcharts even assume it is loaded before the first cycle of the current instruction with special values (such as operand length count, masks, and immediate data).

SH/SL (Micro/370) The 64-bit shifter. It does single and double shifts from 0 to 63 bits in one cycle. SH is the high-order half and is connected only to the B bus. SL is the low-order half and is connected only to the A bus.

Sheet resistivity The electrical resistance per unit area of a substance. If the thickness is assumed fixed, the resistance across a square sheet is the same per square micron or per square mile.

Sign-extend Extend a short right-justified operand by copying the sign bit to fill the most significant (empty) bit position.

Silicon A chemical element with four electrons in its outer shell. Its pure crystalline form is an insulator. If it contains impurities (a few atoms per billion) with three (p-type) or five (n-type) electrons in the outer shell, it becomes a semiconductor.

Simulation Something you have to do to find errors. For Micro/370, we used four kinds of simulation: flowchart, logic, circuit, and switch-level.

Single-chip microprocessor A chip consisting of an execution unit, control store (with all microcode resident), control word decoders, clock, bus controller (for the external bus connection), state sequencer, and instruction decoders. To me, "single-chip" is redundant.

Site (transistor) In silicon gate MOS technology, the intersection of poly and diffusion can be a transistor, so each such intersection is a transistor site. It is one measure of logic density. In a ROM, for example, there may be 1,024 bits in the ROM, but only 400 transistors (the others being not imple-

mented, so they look like zeros). There are 400 "actual" transistors and 1,024 transistor sites. (Compare *Actual [transistors].*)

Source The origin of a signal, such as a register.

Specification A detailed description of function, requirements, contents, components, or whatever.

SSI Small scale integration. Describing an IC logic chip with fewer than about one hundred transistors.

State (Flowcharts) The information in a flowchart box. In a microcoded implementation, a state corresponds to one control word.

State sequencer The logic and registers responsible for sequencing the execution unit through the flowcharts (microcode).

Static (Of circuits) Yielding valid output values as long as the input remains unchanged.

Storage (IBM) Memory.

Synchronous (Hardware) Synchronized with the system clock. An event is synchronous if its occurrence depends only on a clock. Like leaving for lunch at noon sharp, no matter what (as opposed to saying, "We'll leave when she gets to my office.") (Software) Synchronized with the execution of an instruction. An event is synchronous if an instruction must wait for the event to finish before the instruction will complete.

System/370 The name of one IBM computer architecture. The *IBM System/370 Principles of Operation*, manual no. GA22-7000, is the architecture manual.

Task (Flowcharts) A source-to-bus-to-destination transfer.

Technology A specific chip manufacturing process (such as nMOS, pMOS, CMOS, and bipolar) with details (ground rules) of how transistors, resistors, and capacitors are implemented (wire widths, oxide thickness, channel length, doping, sheet resistivity, gate capacitance, and so on).

Testability (Tredennick) A reason used by testing fanatics for things they want in the chip's design. In practice, anything that a testing fanatic says logic and circuit designers should consider.

Test circuits (Tredennick) Things you put on the chip to fill the empty spaces.

Testing There are two kinds. One is the kind you do to find out whether the first chips work. This kind is extensive and includes tests that diagnose problem symptoms. The other is the

kind you need to check the production chips before you try to sell them. These are called manufacturing tests; they have to run fast (otherwise they hold up the production line).

TLB Translation lookaside buffer. A special-purpose cache memory, accessed by virtual addresses, that contains corresponding real addresses. Thus, an address cache.

Transistor The direct electrical embodiment of a spring-loaded valve (normally on or normally off). A transistor is a valve structure. Transistors (electrical valves) come in two types, bipolar and FET (field-effect transistor). That is, they come as bipolar valves and field-effect valves; the two types operate on different electrical principles.

Transistor counts (Sites) The number of places the gate layer crosses the diffusion layer. (A measure of potential logic density for a technology.)
(Actual) The number of real transistors there are. (Useful for simulation and power dissipation calculations.)

Tristate An output that can be logic 1, logic 0, or high impedance (pretends it's not there).

TTL Transistor-transistor logic. SSI logic family 54/74 series part numbers.

Verification (Tredennick) In practice, something you have to claim to have done to get someone above you in the organization to "bless" your design as correct.
(Verification and testing fanatics) Distinguished from testing. Testing aims to see what something does—that is, how it performs as built. Verification aims to see whether the design is correct (meets a functional specification).

VLSI Very large scale integration. Describing an IC logic chip containing more than ten thousand transistors (sites). A current acronym that increases book sales and job title appeal but has no firm definition. Used with other imprecise words such as advanced, design, system, manager, and architecture. Also, chips.

Wafer A thin, polished slice of a silicon crystal. Wafers are circular and 3 to 6 inches in diameter (1986).

Word (IBM System/370) 32 bits. In memory, a word is located on a word (divisible by 4) boundary.

Index

ABC (address bus control), 322
Access exceptions, 142, 146–147
Access indicators, 134–135
Access request, incomplete, 95
Access type description, 44
Access width, 46–47
Add (A) instruction
  decimal, 118, 182–183
  operation tasks for, 30, 31
  overlapping, 81–83
  pretesting, 119
  register-to-memory, 25–27, 29
  register-to-register, 25–26, 29, 33, 52
  with split cycles, 85–87
  steps in, 10–12
Address bus, 305
Address bus control (ABC), 322
Address generator (AG), 129, 326
Address mode sequences, 34–36
  defined, 34
  instruction decoders and, 55–56, 59–61
  sequence labels for, 43
  sharing, 35, 47
Address out (AO) buffer, 127, 194, 195–196, 325
Address space separation, 314
Address strobe, 306
Address temporary (AT) register, 128, 326
Address types, in Micro/370, 271
AG (address generator), 129, 326
Algorithms, 118
ALU (arithmetic and logic unit), 129, 26, 326
  condition codes, 46, 113, 159
  four-phase clock and, 84–85
  inputs to, 24
  in level 1 flowcharts, 36–38
  selecting, 110–111
ALU and condition code (CC) control, 46, 159
ALU constants, 120–121
ALU controls, 71–72
ALU functions, 46, 110–111
AO (address out), 127, 194, 195–196, 325
Architecture, 269–274
Arithmetic and logic unit. *See* ALU
Arithmetic overflow, 92–93
Arithmetic unit (AU), 128, 326

AT (address temporary) register, 128, 326
AU (arithmetic unit), 128, 326
Automated design systems, 235–236
Auxiliary reset, 310

BAL, BALR, 272
BAS, BASR, 272
BC (branch conditionally), 46
Bit patterns
  for control word register control field, 69–71
  for instruction decoders, 61
Bit-slice culture, 245, 249
Block diagram, 56, 256
Branch addresses, 101, 272
Branch and Link (BAL, BALR) instructions, 272
Branch and Save (BAS, BASR) instructions, 272
Branch conditions, 46, 74, 193
Branch control unit, 13, 72–75
Branch instructions, prefetching and, 140–142
Branch microword format, 193
Branch Negative (BN), 74
Branch on Carry (BC), 74
Branch on Condition Register (BCR), 160–161
Branch on Overflow (BV), 74
Branch on Zero (BZ) instruction, 38–39, 74
BSS, 130, 327
Bus
  data, 126, 305, 322
  disadvantages of, 242
Bus controller, 8, 15, 24
  communication with state sequencer, 116–117
  external, 123–124
  external interrupts and, 93
  internal, 122–123, 322–324
  Micro/370, 126, 133–140
  prefetching and, 142
Bus couplers, 127, 129
Bus cycles, 138–140, 311
Bus error, 95–96, 99, 308
Bus general constants, 121–122
Bus grant, 308
Bus grant acknowledge, 308
Bus interface, 123–124, 133–140
  external bus control, 123–124
  synchronization, 123
Bus request, 308
Bus sense amp control, 148–151
Bus status register (BSR), 116
Bus transfers, 57, 59
Byte transfers, 57, 59

Cache memory, 242
CAD, for flowcharts, 243
Calculate Effective Address instruction, 125
Central processing unit (CPU)
  behavior of, 76
  control part of, 5
  data part of, 5
  designing, 16
  notation for, 27–28
  operation of, 78
  speed of, 23
  System/370, 247–248, 274
Chip layout, 87–88, 231–232, 240, 256
Chips
  clocking scheme, 85
  design methods, 221–237
  gate array, 223
  global layout and timing, 87–91
  logic replacement method, 232
  single, 5–6, 224–225, 250, 251–253
  spaghetti, 226
  speed of, 85
Circuit design and layout, Micro/370, 258
CLCL (Compare Logical Long), 272–273
CLM (Compare Logical characters under Mask), 151
Clock, 84–85, 310
  four-phase, 84–85, 90, 148, 149
  phases, 13, 83, 84
Clocking and timing, 148
Combinational controllers, 50, 54–55, 230–231
Combinatorial controller. *See* Combinational controller
Commercial microprocesser design method, 225–231, 232–236

Commercial processor culture, 245, 247–249
Compare Logical characters under Mask (CLM), 151
Compare Logical Long instruction (CLCL), 272–273
Condition codes, 113
  bits, 114
  setting, 46
  in status register, 114
  storing, 116
  Z, 38
Conditional branch instructions, 38–39, 46, 51, 74
  interrupts and, 98
  invisible, 39
  visible, 39
Control code, 271
Control fields
  bit allocation for, 62–63
  bit pattern, 69–71
  sharing, 62
Controller
  algorithms and, 118
  design feedback, 34–36
  efficiency of, 28
  function of, 16
  Micro/370, 126
  notation and, 27
  operation of, 50
  PLA implementation and, 227–228
  requirements, 36
  size of, 23
  System/370, 248
  task merging and, 31
Control lines, 51
  sf, 209
  for a specific register, 67–68
Control logic, 242
Control mode, 124
Control points, 15
Control space, 270
Control store, 8, 14, 27
  bit patterns, 52–53
  branching, 100–101
  controller design and, 53–54
  layout and, 87–88
  Micro/370, 105, 130–133
  operation, 104
  organization, 104–105
  personalizing, 52–53
  size of, 117
  slowness of, 242, 243
  stacking, 194
  System/370, 248
Control store address, 10, 50, 53, 61, 130
  modifier, 51
  next address (NA) field, 51
  selection, 100–103
  starting, 87–88
Control store placement program, 257
Control store word format, 50
Control word decoder, 8, 14–15, 51
  implementation of, 62–72
  in microcoded controller, 56
  timing of, 90–91
Control word format, 76
  implementation of, 61–62
  in microcoded controller, 56
Control words, 8, 14–15, 25, 27, 50–51
  branch control unit and, 74–75
  branches, 100–101
  defined, 260
  microprocessor layout and, 88
  microprocessor operation and, 80
  number in control store, 132–133
  operation section, 61–62
  in register-to-register ADD instruction, 52
  states and, 52
  state section, 61–62
Control word sequence, 8, 10–12, 25, 50, 260
  ALU function control and, 110–111
  bus error interrupt, 95–96, 99
  defined, 34
  instruction decoders and, 59–61
  loading IRE with, 36
  power-on, 96–97
  reset, 96
  storage and, 117
  trace and, 94–95
Convert to Binary (CVB), 177–184
Convert to Decimal (CVD), 180, 184

Conway, Lynn, 232
Coprocessor
  decimal, 167–172
  support, 316
CPU. *See* Central processing unit
Cray, Seymour, 237
CVB, 177–184
CVD, 180, 184

DA (design automation), 2–3
Data bus, 305, 322
  external, 126
Data bus control (DBC), 322
Data input (DI) register, 129, 326
Data output (DO) buffer, 129, 326
Data registers, 66
Data strobes, 138
Data temporary (DT) register, 129, 326
Data transfer acknowledge, 308
Data transfer exception, 308
DBC (data bus control), 322
Decimal addition, 118, 182–183
Decimal coprocessor, 167–172
Decimal instructions, 167–176
Decimal numbers, converting to binary, 177–184
Decimal subtraction, 118, 183–184
Decoders, 133
Decoding, 78–83, 108–113, 140–147
Deferred interrupts, 93, 97–98
  external, 98, 99
  internal, 93, 97–98, 99
Design automation (DA), 2–3
Design chronology, 57, 255–268
Design methods, 221–237
Destination loads, 24
Device attachment, Micro/370, 311–313
DI (data input) register, 129, 326
Diagnose Dual Mode instruction, 316–317
Diagnose instruction, 125, 269, 270, 273, 274, 302, 317
Differential sense amplifiers, 149–150
Direct address microword format, 193
Direct branches, 51
Displacement, 87
Divide-by-zero, 91, 93
Divide instruction, 187–189
DO (data output) buffer, 129, 326
DT (data temporary) register, 129, 326
Dual mode (Micro/370), 124, 270, 271, 302, 316–317
  enable, 311
  operation exception, 272
Duplicate states, 46, 47–49
Dynamic information, 52
Dynamic microprogrammming, 250

EDB (external data bus), 126
EDIN, 55
Edit and Mark instruction (EDMK), 273
Enable, 309, 311
EUhold (execution unit hold), 146
E-unit. *See* Execution unit
Exceptional termination, 142
Exception conditions, 142–143
Exceptions
  microprocessor handling of, 91–100
  operation, 102
  privilege, 103
  specification, 102–103
Execute (EX) register, 128, 325
Execution
  microprocessor operation and, 78–83
  overlap, 140, 142
  pipelined, 242, 243
  speed, 28–29, 85–87
Execution sequence, 30, 34–36
  defined, 34
  instruction decoders and, 59–61
  sequence labels for, 43
  sharing, 36
Execution unit, 8
  block diagram for, 18–19
  bus error interrupt and, 96
  choosing initial, 24
  commercial processor culture and, 248
  communication with state sequencer, 114–116
  decode, 227–228

design, 19, 33–34
efficiency of, 28–29, 33–34
elements, 8
function of, 16
implementation of, 57–59
in microcoded controller, 55
macros, 190–191
merging and, 31
Micro/370, 126, 127–130, 255–256, 322–324
microprogrammable processor culture and, 249
PLA implementation and, 227–228
rules of operation, 18–19, 34
simple, 34
System/370, 247–248
timing of, 90–91
Execution unit control
control word format and, 61–62
implementation of, 194–220
latch, 91
point, 15
Explicit shifts, 151
EX (execute) register, 128, 325
External bus control, 123–124
External bus cycle, 27
External bus protocols, 8
External data bus (EDB), 126
External interrupts, 91–94, 97–98
External read/write line, 193

Fetch instruction, 28–30, 78–83
Finish cycles, 137
FLASH, 258
Floor plan, 231–232, 240, 256
Flowchart assembler, 256, 257
Flowchart Method, 3–4, 16
advantages of, 230, 260
commercial microprocessor design and, 227, 228–231
overview of, 19–22, 259–260
for VLSI design, 228–231
Flowcharts, 16–49
CAD for, 243
CPU depiction by, 27–28
creating, 25–27
defined, 17
design objectives, 23–24
execution speed, 28–29
hardware and, 55–56
implementing from, 50–76
implementing Micro/370 from, 190–220
level 1, 30–32, 36–39, 260
level 2, 25, 32–36, 39–49, 156–158, 260
materials for, 157
for Micro/370, 155–189, 190–220, 275–300
notation, 27–28
objects, 23–24
prerequisites for, 18–19
simulation, 256–258
syntax checker, 257
System/370, 256
uses of, 17
writing, 25–28
Folded PLAs, 205–208, 213–220. *See* also PLA
disadvantages of, 219–220
Four-phase clock, 84–85, 90, 148, 149
FRIN, 137
FROP, 137
Function codes, 137, 138, 193, 305–306, 314

Gate array, 223
Gate control signal, 56
General registers, 109–110, 124–125, 129, 326
Global chip layout and timing, 87–91
GPR (general and shadow) registers, 129, 326. *See also* General registers; Shadow registers

Hadsell, Dick, 158, 236, 257
Halfant, Matt, 255
Halfword
microprocessor operation and, 79–83
overlapping and, 81–83
transfers, 57
HALT, 140, 310
Hardware flowcharts. *See* Flowcharts
Hardwired controller. *See* Combinational controller
Hexadecimal conversion, 179–181

Housekeeping tasks, 260
in level 1 flowcharts, 30–31
merging with operation tasks, 30–31, 39–41

IB (instruction branch), 46, 59–61
IBM Micro/370. *See* Micro/370
IBM System/370. *See* System/370
IC (Insert Character), 122
ICM (Insert Characters under Mask), 122, 151
ILC (instruction length count), 130, 327
Immediate interrupts, 93, 94, 97–98
external, 94, 98, 99
internal, 93, 99
Implicit shifts, 151
Input clock, 310
Insert Character (IC), 122
Insert Characters under Mask (ICM), 122, 151
Instruction, 10–15
bit pattern, 55
execution process, 36
execution time, 85–87
fetch, 28, 29, 30
function codes, 137
length, 34
mnemonics, 61
Instruction branch (IB), 46, 59–61
Instruction decoder, 8, 10, 36, 101–103
implementation of, 59–61
Micro/370, 133
in microcoded controller, 55–56
output of, 87–88
PLA implementation and, 227–228
sequence labels and, 43
timing of, 90
Instruction length count (ILC), 130, 327
Instruction register for execution (IRE). *See* IRE
Instruction register for fetch (IRF). *See* IRF
Instruction registers, 116
Instruction set summary, 18
Instruction tracing, 94
Intel 8086, 253
Interface control unit (ICU), 311–312
Internal bus control, 122–123
Internal buses, 322–324
Internal clocking, 84–85
Internal interrupts, 91–94, 97–98
Interruptions, 75–76, 274
pending, 144–145
Interrupts, 13–14, 91–99
acknowledge cycles, 138
conditional branching and, 98
deferred, 93, 97–98
external, 91–94, 97–98
immediate, 93, 94, 97–98
internal, 91–94, 97–98
mask bits, 114–115
multiple, 99–100
priority level, 308
ranking, 99–100
IRD (instruction register for decode), 116, 130, 327
IRE (instruction register for execution), 32, 33, 50, 130, 327
loading, 36
microprocessor layout and, 89
static information in, 52
timing of, 91
IRF (instruction register for fetch), 32, 116, 130, 327
IRIN, 137
IROP, 137

Karnaugh maps, 62, 63, 64, 69, 72, 86, 136, 208–209

Lam, Linh, 256
Large scale integration (LSI), 221
Layout. *See* Chip layout
Level 1 flowcharts, 30–32, 260
controller design, 34
creating, 36–39
for Micro/370, 158
Level 2 flowcharts, 32–36, 260
adding descriptive information to, 42–47
creating, 39–49
execution unit design, 33
official, for Micro/370, 158–160
practice, for Micro/370, 156–158
Load Address (LA) instruction, 122, 273

Loan Multiple (LM) instruction, 109, 161–163
Logic design, 57, 258
Logic replacement design method, 231–236
LDS/SA1, 306

Machine partition design method, 222–225, 232–236
Macro catalog, 222–223
Macros, 76
  design of, 159
  internal bus control and, 122–123
  in machine partition design, 222–224
  selection of, 155
  uses for, 63
MC68000. *See* Motorola MC68000
Mead, Carver, 232
Memory
  access bus codes, 137–138
  access cycles, 314
  addresses, 271
  internal clocking and, 84–85
Merging tasks, 30–31
Micro/292, 170–171
Micro/370, 1, 13–15, 126–154, 301–327
  addresses, 271
  vs. alternatives, 242–243
  ALU constants, 120
  architecture notes, 269–274
  bus controller, 122, 123, 126, 133–140
  bus general constants, 121–122
  bus sense amp control, 148–151
  clocking and timing, 148
  component description, 302–304
  condition codes, 113
  controller, 17, 126
  control store, 105, 130–133
  control word format, 62
  design, 260–268, 314–318
  device attachment, 311–313
  execution overlap, 140
  execution unit, 19, 57–59, 126, 127–130, 255–256, 322–324
  external data bus (EDB), 126
  flowcharts, 155–189, 190–220, 275–300
  instruction decoders, 102, 133
  interrupt ranking in, 99
  layout, 240, 258
  logic design, 57, 258
  managers and staff, 263
  mode control in, 124–125
  next address control, 133
  next state control for, 75
  overview of, 302
  package information, 305
  prefetching, 140–148
  processor states, 318–321
  register control in, 109–110
  shifter and shifter control, 151–154
  signal descriptions, 305–311
  special function unit, 325–327
  states, 118
  status register in, 114–115
  System/370 instructions not implemented on-chip, 270
  technical contributors, 262–263
  trace in, 94
Microcode
  branches, 132–133
  compaction, 240, 251–252
  process, 8, 240
  state sequencer and, 117–118
Microcoded controller
  design of, 53–54
  flowchart implementation, 50
  sharing address mode sequences in, 36
  steps for implementing from flowchart, 55–56
Microcoded processor
  implementation, 6–8
  operational overview of, 8
  VLSI designs, 231
Micro control store, 130
Microcycle, 26, 85–87, 90
Microprocessors
  communication paths for, 87
  defining, 5–15
  operation of, 8, 78–123
  overlapping operations in, 78, 80–83
  single-chip, 5–6, 224–225
Microprogrammable processors, 248
  culture, 245, 246, 249–250

Microprogrammed processors, 250, 253
Microprogramming, 247
  alternative styles, 242–243
  "cultures" of, 245–254
  changes in, 252
  dynamic, 250
  vs. programming, 251
  single-chip culture and, 250, 251–253
Microword formats, 190
  branch, 193
  direct, 193
Microwords, 104, 130–133
  control store, 194
  defining, 193–194
  next address control and, 133
  placement, 252
MIN processor
  control word decoders, 62–72
  descriptive flowchart information for, 42–49
  execution unit, 25–26
  implementation, 50, 57–72
  instruction decoders, 55, 59
  next state control for, 75–76
  registers, 66
  state sequencer, 72–76
Mnemonic control store addresses, 61
Mode bits, 115, 124
Mode control, 124–125
Motorola MC68000, 8, 12, 13–15, 34, 47
  address modes, 101
  ALU constants, 120
  bus general constants, 122
  condition codes and, 113
  controller for, 17
  control store, 104–105, 132, 194, 253
  control words in, 52
  device attachment, 311–313
  instruction decoders, 102
  instruction registers in, 116
  internal bus control, 122
  logic design steps, 57
  Micro/370 compatibility with, 269–270, 303
  single-chip culture and, 250–251
  speed of, 253
  states in, 118
  status register in, 114–115
  trace in, 94–95
Move Character Long (MVCL) instruction, 113, 273–274
Multiple interrupts, 99–100
Multiplexers, 104–105
Multiply instruction, 74, 108–109, 184–187
  algorithms for, 118
  conditional branches and, 39
MVCL (Move Character Long) instruction, 113, 273–274

NACC, 140
Nano control store, 130
Nanodata QM1, 250
Nanoword decoder, 105–113
  for ALU constants, 121
  timing and, 107
Nanowords, 104, 130–132
  bit patterns, 191–193
  control field, 196
  control store, 194
  defining, 191–193
  encoding, 105–106
  fields, 105–106
  format, 190, 191
Next address control, 133
Next control store address, 75–76
Next instruction fetch, 30
Next state control, 75–76
Next state transition, 46
Notation, 25, 27–28

Op code, 87
Operands, 10, 26
  function codes, 137–138
  length fields, 116
  in Micro/370, 272
Operation exceptions, 92, 102
Operation tasks, 260
  in level 1 flowcharts, 30–31
  merging with housekeeping tasks, 30–31, 39–41
Overlapping, 78, 80–83, 140, 142

Pack instruction, 196–197
Pack-Unpack Register (PUR). *See* PUR

Page and location coordinates, 46
PC (program counter), 128
  examples, 63–64
  increment, 28, 29, 30
  register, 325
  relative address modes, 41–42
  update, 41–42
PCR (processor command register), 117, 134
Pending interruptions, 144–145
Pipelining, 242
PLA (programmable logic array), 6–8, 191
  advantages of, 227
  commercial microprocessor design, 227–228, 231
  conventional, 212
  conventional PUR, 200–208
  conventional, folded, 213–220
  folded, 205–208, 219–220
  PUR, 196–200
  SFU, 209–212
  SFU conventional, 212–220
PLA controllers
  design, 53–55
  flowchart implementation, 50
Power-on, 13, 96–97
Precharging, 150
Prefetching, 15, 32, 140–148
Pretesting, 118–119
Priority request, 309
Privilege exceptions, 91, 103, 272
Processor command register (PCR), 117, 134
Program counter. *See* PC
Programmable logic array. *See* PLA
Programming, vs. microprogramming, 251
Program status word (PSW)
  swap, 94, 271, 274
PSW (Program status word), 94, 271, 274
PSWH, 130, 327
PSWL register, 128, 325
PUR (Pack-Unpack Register), 129, 194, 326
  conventional PLA for, 200–208
  example, 196–200
  folded PLA, 205–208
RAM control store, 248
Random logic, 6–8
  vs. microprogrammed processors, 249
  vs. PLA, 227–228
  speed of, 253
Random logic controller. *See* Combinational controller
RCS (reloadable control store), 248
Read, 27
  function codes, 137–138
Read/write, 253, 306
Real address tag, 305
Register
  designators, 52
  general, 109–110, 124–125, 129, 326
  interconnect, 242
  logical grouping of, 66
  physical grouping of, 66
  pointers, 66, 88–89
  selection, 67
  shadow, 110, 124–125, 129, 316–317, 326
  transfer notation, 25, 27–28
Register control
  decoding and, 108–110
  examples, 65–71
Register-to-memory ADD instruction, 25–27, 29
Register-to-register ADD instruction, 25–26, 29, 33, 52
Reloadable control store (RCS), 248
Requestor, for trace, 94
Reset, 75–76, 96, 310
REST, 140
ROM, 4, 104–105, 134
Rules-of-operation timing, 26
RX, 65–71
RY, 65–71

SB (sequence branch), 46, 59–61
Select read cycle, 311
Sense amps, 129, 148–151, 324
Sequence branch (SB), 46, 59–61
Sequence labels, 43–45
Sequence sharing, 35–36
Serial operation, 78

Service cycles, 138–140, 170–171, 314
SFU (special function unit), 116, 129, 194, 208–220, 327
  conventional PLA, 212–220
  Micro/370, 325–327
  PLA, 209–222
SH/SL, 326
SH/SL/OVF, 129
Shadow registers, 110, 124–125, 129, 316–317, 326
Shifter and shifter control, 151–154, 187, 326
Shifts
  explicit, 151
  implicit, 151
Shimamoto, Brion, 137, 157, 250, 255
Signal descriptions, 305–311
Silicon compilers, 236
Simulation, 256–258
Single-chip culture, 245, 246, 250, 251–253
Single-chip microprocessor, 5–6. *See also* Microprocessor
  defined, 250
  design, 224–225
Spaghetti chips, 226
Special function unit (SFU). *See* SFU
Specification exceptions, 102–103
Split cycles, 85–86
SRAK, 138
SRMC, 139
SRPG, 139
Stack pointers, 66
State identification (state ID), 46, 47
States, 12, 17, 24, 25
  control words for, 52
  defined, 259
  duplicate, 46, 47–49
  identifying, 42
  in level 2 flowcharts, 158
  Micro/370, 318–321
  number of, 30–31, 41, 47–49, 117–18
State sequencer, 8, 14, 27, 72–76
  access exceptions and, 142
  branch control unit, 72–75
  bus controller interface and, 133–134
  communication with bus controller, 116–117
  communication with execution unit, 114
  control word format and, 61–62
  defined, 259
  next state control, 75–76
  prefetching and, 142–147
Static decoding, 106
Static information, 51–52
Status register (SR), 114–116, 142
STM (Store Multiple) instruction, 109
Store Characters under Mask (STCM) instruction, pretesting, 119
Store instruction, 38
Store Multiple (STM) instruction, 109
Subroutines, 35
Subtraction, decimal, 118, 183–184
Suppress fullword write, 311
SWAK, 139
SWMC, 139
SWPG, 139–140
Synchronization, 123
Synonym, 47
System/360 Model 91, 249
System/370
  address types, 271
  bus general constants, 121
  commercial processor culture and, 247–249
  compatibility of Micro/370 with, 269
  condition codes and, 113
  CPU, 247–248, 274
  instructions, 160–189
  internal bus control, 122
  interruptions, 94, 309
  Model 195, 249
  pretesting, 118–119
  register control in, 108–109
  state indicators, 310
  Status Register in, 114–115
  trial execution, 119–120
System reset signals, 75–76

T2 control
  examples, 64–65
Tasks, 17, 25, 27
  defined, 259
  housekeeping, 30–31, 260
  notation and, 28
  operation, 30–31, 260
  rules-of-operation timing for, 26
Time, in flowchart, 26
Timing
  between execution unit and external bus, 85–87
  nanoword decoder and, 107
  signals, 51
  for whole chip, 87–91
Total cycles, 137
TR (Translate) instruction, 119–120, 163–167
Trace, 91, 93, 94–95, 99
Transfers, 27, 43
Translate (TR) instruction, 119–120, 163–167
Translate and Test (TRT) instruction, 109, 273–274
Trial execution, System/370, 119–120
TRIN, 137
TROP, 137
TROW, 137–138
TRRO, 138
TRT (Translate and Test) instruction, 109, 273–274
Tsai, Mon Yen, 155, 199, 256, 258

UDS/SA2, 306

Valid memory address/output service strobe, 309–310
Valid peripheral address, 309
Very large-scale integration (VLSI). *See* VLSI design
Virtual storage support, 317
VLSI design
  chronology, 255–268
  combinational, 230–231
  commercial microprocessor method, 225–231, 232–236
  costs of, 235
  directions in, 221–222
  flowcharts for, 228–231
  logic replacement method, 231–236
  machine partition method, 222–225, 232–236
  methods, 221–237
  microcoded, 231
  miracle solutions for, 235–237
VMA/OSS, 309–310
VPA, 309

WS/SA3, 306
Word transfers, 57, 59, 308
Write, 27
  function codes, 137–138
WROP, 138
WRRO, 138

Z condition code, 38
Zilog Z8000, 253

Other Titles from Digital Press

VAX Architecture Reference Manual. Edited by Timothy E. Leonard. Authoritative, comprehensive, and precise description of the data types, instructions, calling standards, addressing modes, registers, exception and interrupt handling, memory management, and process structure common to all VAX computers—from the Micro-VAX II to the VAX 8800.

VAX/VMS Internals and Data Structures. Lawrence J. Kenah and Simon F. Bate. A complete explanation of how the VAX/VMS executive works.

Computer Programming and Architecture: The VAX-11. Henry M. Levy and Richard H. Eckhouse, Jr. Discusses all aspects of VAX programming and organization.

Engineering a Compiler: VAX-11 Code Generation and Optimization. Patricia Anklam, David Cutler, Roger Heinen, Jr., M. Donald MacLaren.

Parallel Processing: The CM* Experience. Edward F. Gehringer, Daniel P. Siewiorek, Zary Segall.

Capability-Based Computer Systems. Henry M. Levy.

The Theory and Practice of Reliable System Design. Daniel P. Siewiorek, Robert S. Swarz.

Computer Engineering: A DEC View of Hardware Systems Design. C. Gordon Bell, J. Craig Mudge, John E. McNamara.

For a complete Digital Press catalog write:
Sales Manager
Digital Press
Digital Equipment Corporation
12 Crosby Drive BU0/E94
Bedford, MA 01730

"This book is a refreshing change from most computer engineering textbooks. It gives insight into what it takes to design a commercial-grade microprocessor, not just some oversimplified, paper design. It is also the first book I've seen that describes microprogramming as it is really practiced, where the microarchitecture is carefully tailored to the macroarchitecture."

Randal E. Bryant, Carnegie-Mellon University

digital

Digital Press
Digital Equipment Corporation
12 Crosby Drive
Bedford, Massachusetts 01730

Order Number EY-6707E-DP
ISBN 0-932376-92-4